University of California
Los Angeles

Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Peter Vincent Homeier

1995

The dissertation of Peter Vincent Homeier is approved.

Rajive Bagrodia

Donald A. Martin

D. Stott Parker

David F. Martin, Committee Chair

University of California, Los Angeles

1995

I dedicate this dissertation

to the most wonderful friend I have ever had,

my Messiah, Lord, and Savior, the Son of God,

Y'shua Ha Mashiach, Jesus the Christ.

He has cared for every need faithfully,

in the midst of earthquake and opposition.

He has wholeheartedly poured out His Holy Spirit on me.

At each point of difficulty, at each resistant problem,

like a cool drop of rain, He quietly dropped the answer into me.

When the ultimate impossible cliff arose before me,

He opened doors of understanding,

drawing me beyond what I thought was the end,

through the darkness of the grave

to the dawn of a new morning.

He has been the lifter of my head, and the restorer of my hopes.

Of all the people I know,

He is the most precious to me.

He loved me enough to humbly go to the Cross and die in my place.

I can never repay such a pure and shattering gift.

This dissertation is only the smallest of ways

I can express my heart's wonder and love

for such a greater love He has lavished on me.

TABLE OF CONTENTS

List of Figures

List of Tables

Acknowledgments

My earnest thanks go to Professors Rajive Bagrodia, Donald A. Martin, D. Stott Parker, and my advisor, David F. Martin, for serving on my committee.

I am grateful to The Aerospace Corporation for supporting my education with a fellowship for several years through most of my graduate studies.

Ching-Tsun Chou read early drafts of this manuscript and provided many helpful comments and suggestions. We have faced the heat of the battle together, and I have enormous respect for his intellect. I am grateful to Raymond Toal for his cheerful inspiration and solid example.

I thank my wonderful father and stepmother, Skip and Della Homeier, for too much to say, but especially for their encouragement to return to school and pursue the Ph.D. Their words sparked the desire of my heart to become a manifest reality, and have profoundly sharpened my life as a sword on the anvil.

Pastor Jim Nelson at The Church on the Way has given me consistent and faithful support over the years. From the beginning of this quest, with insight and devotion he has tenaciously watched over me, and I shall always be grateful for his protective care. I also want to thank Pastor Chip Graves, for praying and encouraging me when I needed to choose between my job and my degree.

Pastor Jack Hayford has been a staff and a covering for me. His wisdom and simple sincerity of heart have been a living model, blending biblical brilliance with tender faith and honest humility. He has an uncanny ability to speak to the heart of a situation, inevitably displaying the perspective of heaven. He has

focused me on the essentials, on integrity of heart, passion for fullness, and fierceness of commitment, as an elite trained soldier. (Isaiah 28:5-6, Zechariah 6:11, 1 Peter 5:4) The little boy who was healed from falling down is now teaching us all to go “leaping upon the mountains, skipping upon the hills.” My respect and love for him know no bounds.

Verra Morgan is a shining angel, clucking like a mother hen. Her fierce, eager, combative spirit is always ready like a growling mother bear to defend her cubs, and then to cuff them when they misbehave. She has saved my neck administratively more times than I can count. I honestly do not know if I would have completed my degree if in the accident she had been taken from us. My heart overflows with thanks she was not. Verra has always lived for the success of the graduate students in her care. May she someday find a joy even greater.

Susan has been a gift beyond expectations, knitting raveled edges. She has shown me a mother’s love, faith, and encouragement in the secret depths, with wisdom that few have ever seen. “For the Spirit searches all things, yes, the deep things of God. For what man knows the things of a man except the spirit of the man which is in him?” (1 Corinthians 2:10–11) She has eyes for the light in the heart of the dark, and a rod of healing held over troubled waters. My deep thanks go to her for our many rich words and woven times.

My special heartfelt appreciation goes to my advisor, Professor David F. Martin, for his supervision of this research. He has been a guiding light, an inspiration, a constant source of encouragement and approval, and the warmest of friends. He has truly been “the advisor from Heaven,” in every way going beyond all that I ever conceived or could have asked for. I want to always give the same conscientious and diligent care he has given me. Through this long walk together, he has become a father to me, cherished beyond words. May his brilliance and his heart be recognized and rewarded.

My highest gratitude goes to someone not seen on Earth, who spun the galaxies with His fingertips, and blew the stars aflame. It was He who gave me the strength to leap each wall. Indeed every good and perfect gift comes down from above, from the Father of Lights. When opposition arose demanding I stop, He provided the way to continue and grow. When I was at the end of myself, with all my hopes at risk, He reached down and kept me from sliding into the pit, and placed my feet on a rock. He who said, "Let there be light," has shone in my heart. (2 Corinthians 4:6) All of this dissertation is only here by His hand extended in grace and peace.

Soli Deo Gloria.

Vita

1956	Born, Santa Monica, California, USA
1979	B.S. in Mathematics/Computer Science Summa Cum Laude Mathematics/Computer Science Senior Prize University of California, Los Angeles (UCLA) Los Angeles, California, USA
1979	Regents Fellowship University of California, Los Angeles (UCLA) Los Angeles, California, USA
1981	M.S. in Computer Science University of California, Los Angeles (UCLA) Los Angeles, California, USA
1981–1983	Member of the Technical Staff Hughes Aircraft Corporation El Segundo, California, USA
1983–1995	Member of the Technical Staff The Aerospace Corporation El Segundo, California, USA

1988–1994 Aerospace Graduate Fellowship
The Aerospace Corporation
El Segundo, California, USA

Publications

Peter V. Homeier and David F. Martin, "A Mechanically Verified Verification Condition Generator," *The Computer Journal*, Vol. 38, No. 2, 1995, pages 131–141.

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 F. Melham and Juanito Camilleri, Valletta, Malta, September 19–22, 1994, Lecture Notes in Computer Science Vol. 859, Springer–Verlag, pages 269–284.

Peter V. Homeier, Thach C. Le, Y. Peter Li, and Peter C. Eggan, "DEF–CLIPS: Extensions to the CLIPS Production System Environment," in *Proceedings of Artificial Intelligence/Expert Systems Symposium*, El Segundo, CA, September 1–2, 1993.

Peter V. Homeier and Thach C. Le, "ECLIPS: An Extended CLIPS for Backward

Chaining and Goal-Directed Reasoning," in *Proceedings of the Second CLIPS Users Group Conference*, September 23–25, 1991, Houston, Texas, NASA Conference Publication 10085, Volume 2, pages 213–225.

Thach C. Le and Peter V. Homeier, "PORTABLE INFERENCE ENGINE: An Extended CLIPS for Real-Time Production Systems," in *Proceedings of the Second Annual Workshop on Space Operations Automation and Robotics*, (SOAR '88), July 20–23, 1988, NASA Conference Publication 3019, pages 187–192.

Abstract of the Dissertation

Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures

by

Peter Vincent Homeier

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1995

Professor David F. Martin, Chair

As an alternative to testing, formal proofs of a program's correctness may be constructed. The application of these techniques has been limited by the difficulty of constructing the required proofs by hand. The task of proving a program correct can be simplified and eased by a tool called a *Verification Condition Generator* (VCG), which automatically constructs a significant portion of the proof. The VCG processes programs written in the specified language, and produces as its result a set of lemmas called *verification conditions*, as the remainder left for the programmer to prove. The truth of these is intended to imply the correctness of the program. However, most VCGs that have been written have not themselves been verified, making that support unreliable.

We have written a VCG and verified its soundness, proving that if the verification conditions produced are true, then the original program is totally correct. This proof is conducted within and checked by the Higher Order Logic (HOL) mechanical proof checker, ensuring its complete soundness. The resulting verified VCG provides an effective means for proving programs totally correct with complete security.

The programming language studied contains two areas of special interest, expressions with side effects, and mutually recursive procedures, with global variables and variable and value parameters. As part of this work, we provide five program logics which together provide an axiomatic semantics for total correctness. Of the five program logics, three are fundamental inventions in this dissertation. These new logics are used to verify the correctness of expressions, the progress achieved between recursive calls of the same procedure, and the termination of procedures. All of these logics are mechanically proven within the HOL system to be sound with respect to the formal semantics of the language.

The most novel contribution of this dissertation is the discovery of a new method for proving the termination of programs with mutually recursive procedures, which is both more general and easier to use than prior proposals. In addition, VCG automation is naturally supported. This method analyzes the structure of the procedure call graph, generating verification conditions based on the cycles found.

Part I

Background

CHAPTER 1

Introduction

"Behold, You desire truth* in the inward parts,

And in the hidden part You will make me to know wisdom."

— Psalm 51:6 [1]

* "**truth,** *'emet* (*eh*-met); Strong's #571: Certainty, stability, truth, rightness, trustworthiness. *'Emet* derives from the verb *'aman*, meaning "to be firm, permanent, and established." *'Emet* conveys a sense of dependability, firmness, and reliability. Truth is therefore something upon which a person may confidently stake his life."

— The Spirit-Filled Life Bible, Thomas Nelson Publishers, 1991, page 774.

Good software is very difficult to produce. This contradicts expectations, for building software requires no large factories or furnaces, ore or acres. It consumes no rare, irreplaceable materials, and generates no irreducible waste. It requires no physical agility or grace, and can be made in any locale.

What good software does require, it demands of the intelligence and character of the person who makes it. These demands include patience, perseverance, care,

[1] All quotations from the Bible are taken from the New King James Version, copyright ©1991 Thomas Nelson, Inc., unless otherwise indicated.

craftsmanship, attention to detail, and a streak of the detective, for hunting down errors. Perhaps most central is an ability to solve problems logically, to resolve incomplete specifications to consistent, effective designs, to translate nebulous descriptions of a program's purpose to definite detailed algorithms. Finally, software remains lifeless and mundane without a well-crafted dose of the artistic and creative.

Large software systems often have many levels of abstraction. Such depth of hierarchical structure implies an enormous burden of understanding. In fact, even the most senior programmers of large software systems cannot possibly know all the details of every part, but rely on others to understand each particular small area.

Given that creating software is a human activity, errors occur. What is surprising is how difficult these errors often are to even detect, let alone isolate, identify, and correct. Software systems typically pass through hundreds of tests of their performance without flaw, only to fail unexpectedly in the field given some unfortunate combination of circumstances. Even the most diligent and faithful applications of rigorous disciplines of testing only mitigate this problem. The core remains, as expressed by Dijkstra: "Program testing can be used to show the presence of bugs, but never to show their absence!" [Dij72] It is a fact that virtually every major software system that is released or sold is, not merely suspected, but in fact *guaranteed* to contain errors.

This degree of unsoundness would be considered unacceptable in most other fields. It is tolerated in software because there is no apparent alternative. The resulting erroneous software is justified as being "good enough," giving correct answers "most of the time," and the occasional collapses of the system are shrugged off as inevitable lapses that must be endured. Virtually every piece of software that is sold for a personal computer contains a disclaimer of *any* particular performance at all. For example, the following is *typical*, not extraordinary:

> "X" CORPORATION PROVIDES THIS SOFTWARE "AS IS" WITHOUT ANY WARRANTEE OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL "X" CORPORATION BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF BUSINESS, LOSS OF USE OR DATA, INTERRUPTION OF BUSINESS, OR FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY KIND, EVEN IF "X" CORPORATION HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES ARISING FROM ANY DEFECT OR ERROR IN THIS SOFTWARE.

The limit to which many companies stand behind their software is to promise to reimburse the customer the price of a floppy disk, if the physical medium is faulty. This means that the customer must hope and pray that the software performs as advertised, for he has no firm assurance at all. This lack of responsibility is not tolerated in most other fields of science or business. It is tolerated here because it is, for all practical purposes, impossible to actually create perfect software of the size and complexity desired, using the current technology of testing to detect errors.

There is a reason why testing is inadequate. Fundamentally, testing examines a piece of software as a "black box," subjecting it to various external stimuli, and observing its responses. These responses are then compared to what the tester expected, and any variation is investigated. Testing depends solely on what is externally visible. This approach treats the piece of software as a mysterious locked chest, impenetrable and opaque to any deeper vision or understanding of its internal behavior. A good tester does examine the software and study its structure in order to design his test cases, so as to test internal paths, and check circumstances around boundary cases. But even with some knowledge of the internal structure, it is very difficult in many cases to list a sufficient set of cases that will exhaustively test all paths through the software, or all combinations of circumstances in which the software will be expected to function.

In truth, though, this behavioral approach is foreign to most real systems in physics. Nearly all physical systems may be understood and analyzed in terms of their component parts. It is far more natural to examine systems in detail, by investigating their internal structure and organization, to watch their internal processes and interrelationships, and to derive from that observation a deep understanding of the "heart" of the system. Here each component may be studied to some degree as an entity unto itself, existing within an environment which is the rest of the system. This is essentially the "divide and conquer" strategy applied to understanding systems, and it has the advantage that the part is usually simpler than the whole. If a particular component is still too complex to permit immediate understanding, it may be itself analyzed as being made up of other smaller pieces, and the process recurses in a natural way.

This concept was recognized by Floyd, Hoare, Dijkstra, and others, beginning about 1969, and an alternative technique to testing is currently in the process of being fashioned by the computing community. This approach is called "program correctness" or "software verification." It is concerned with analyzing a program down to the smallest element, and then synthesizing an understanding of the entire program by composing the behaviors of the individual elements and subsystems. This attention to detail costs a good deal of effort, but it pays off in that the programmer gains a much deeper perception of the program and its behavior, in a way that is complete while being tractable. This deeper examination allows for stronger conclusions to be reached about the software's quality.

As opposed to testing, verification can trace *every* path through a system, and consider *every* possible combination of circumstances, and be certain that nothing has been left out. This is possible because the method relies on mathematical methods of proof to assure the completeness and correctness of every step. What is actually achieved by verification is a mathematical proof that the program being studied satisfies its specification. If the specification is complete and correct, then the program is guaranteed to perform correctly as well.

However, the claims of the benefits of program verification need to be tempered with the realization that substantially what is accomplished may be considered an exercise in redundancy. The proof shows that the specification and the program, two forms of representing the same system, are consistent with each other. But deriving a complete and correct formal specification for a problem from the vague and nuanced words of an English description is a difficult and uncertain process itself. If the formal specification arrived at is not what was truly intended, then the entire proof activity does not accomplish anything of worth. In fact, it may have the negative effect of giving a false sense of certainty to the user's expectations of how the program will perform. It is important, therefore, to remember that what program verification accomplishes is limited in its scope, to proving the consistency of a program with its specification.

But within that scope, program verification becomes more than redundancy when the specification is an abstract, less detailed statement than the program. Usually the specification as given describes only the external behavior of the program. In one sense, the proof maps the external specification down through the structure of the program to the elements that must combine to support each requirement. In another sense, the proof is good engineering, like installing steel reinforcement within a largely concrete structure. The proof spins a single thread through every line of code—but this single thread is far stronger than steel; it has the infinite strength of logical truth. Clearly this greatly increases one's confidence in the finished product. Here is the relevance of the introductory quote from Psalm 51. A system is far stronger if it has internal integrity, rather than simply satisfaction of an external behavioral criterion. The heart of the system must be correct, and to achieve this requires "wisdom" (truth) in the "hidden part."

The theory for creating these proofs of program correctness has been developed and applied to sample programs. It has been found that for even moderately sized programs, the proofs are often long and involved, and full of complex details. This raises the possibility of errors occurring in the proof itself, and brings into question *its* credibility.

This situation naturally calls for automation. Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step, and ensures its soundness. This tool becomes an agent which mechanically verifies the proof's correctness. The Higher Order Logic (HOL) proof assistant is such a mechanical proof checker. It is an interactive theorem-proving environment for higher order logic, built originally at Edinburgh in the 1970's, based on an approach to mechanical theorem proving developed by Robin Milner. It has been used for general theorem proving, hardware verification, and software verification and refinement for a variety of languages. HOL has the central quality that only true theorems may be proved, and is thus secure. It performs only sound logical inferences. A proof is then a properly composed set of instructions on what inferences to make. Each step is thus logically consistent with what was known to be true before. The result of a successful proof is accredited with the status of "theorem," and there is no other way to produce a theorem. The derivation is driven by the human user, who makes use of the facilities of HOL to search and find the proof.

Even greater assistance for program verification may be provided by a tool which writes the proof automatically, either in part or in whole. One kind of mechanical tool that has been built is a *Verification Condition Generator* (VCG). Such a tool analyzes a program and its specification, and based on the structure of the program, constructs a proof of its correctness, modulo a set of lemmas called verification conditions which are left to the programmer to prove. This is a great aid, as it twice reduces the programmer's burden, lessening both the volume of proof and the level of proof. Many details and complexities can be automatically handled by the VCG, and only the essentials left to the programmer. In addition, the verification conditions that remain for him to prove contain no references to programming language phrases, such as assignment statements, loops, or procedures. The verification conditions only describe relationships among the underlying datatypes of the programming language, such as integers, booleans, and lists. All parts of the proof that deal directly with programming language constructs are handled automatically by the VCG. This does not mean that there cannot be depth and difficulty in proving the verification conditions; but the program proving task has been significantly reduced.

Several example Verification Condition Generators have been written by various researchers over the past twenty years. Unfortunately, they have not been enough to encourage a widespread use of program verification techniques. One problem area is the reliability of the VCG itself. The VCG is a program; and just as any other program, it is subject to errors. This is critical, however, because the VCG is the foundation on which all later proof efforts rest. If the VCG is not sound, then even after proving all of the verification conditions it produces, the programmer has no firm assurance that in fact he has proven his original program correct. Just stating a set of rules for proving each construct in a programming language is not enough; there is enough subtlety in the semantics of programming languages to possibly invalidate rules which were arrived at simply by intuition, and this has happened for actual rules that have been proposed in the literature. There is a need for these rules, and the VCGs that incorporate them, to be rigorously proven themselves.

This we have done in this dissertation. We present a verified Verification Condition Generator, which for any input program and specification, produces a list of verification conditions whose truth in fact implies the correctness of the original program with respect to its specification. This verification of the VCG is proven as a theorem, and the proof has been mechanically checked in every detail within HOL, and thus contains no logical errors. The reliability of this VCG is therefore complete.

Program verification holds the promise in theory of enabling the creation of software with qualitatively superior reliability than current techniques. There is the potential to forever eliminate entire categories of errors, protecting against the vast majority of run-time errors. However, program verification has not become widely used in practice, because it is difficult and complex, and requires special training and ability. The techniques and tools that are presented here are still far from being a usable methodology for the everyday verification of general applications. The mathematical sophistication required is high, the proof systems are complex, and the tools are only prototypes. However, the results of this dissertation point the direction to computer support of this difficult process that make it more effective and secure. Another approach than testing is clearly needed. If we are to build larger and deeper structures of software, we need a way to ensure the soundness of our construction, or else, inevitably, the entire edifice will collapse, buried under the weight of its internal inconsistencies and contradictions.

CHAPTER 2

Underlying Technologies

"According to the grace of God which was given to me, as a wise master builder I have laid the foundation."
— 1 Corinthians 3:10

Every building has a foundation. The foundation of this dissertation is the collection of technologies that underlie the work. This chapter will describe these technologies, and give a sense of how these elements fit together to support the goal of program verification.

To make this more concrete, we will take as an example a small programming language, similar to a subset of Pascal, with assignment statements, conditionals, and while loops. Associated with this language is a language of assertions, which describe conditions about states in the computer. For these languages, we will define their syntax and semantics, and give a Hoare logic as an axiomatic semantics for partial correctness. Using this logic, we will define a Verification Condition Generator for this programming language. Finally, we will discuss embedding this programming language and its VCG within Higher Order Logic.

This small programming language is not the language actually studied in

$$
\begin{array}{|lcl|}
\hline
c & ::= & \textbf{skip} \\
 & | & \textbf{abort} \\
 & | & x := e \\
 & | & c_1 \; ; \; c_2 \\
 & | & \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ fi} \\
 & | & \textbf{assert } a \textbf{ while } b \textbf{ do } c \textbf{ od} \\
\hline
\end{array}
$$

Table 2.1: Example programming language.

this dissertation, but in its simplicity serves as a clear illustration to discuss the fundamental technologies and ideas present in this chapter.

2.1 Syntax

Table 2.1 contains the syntax of a small programming language, defined using Backus–Naur Form as a context–free grammar. We denote the type of commands as `cmd`, with typical member c. We take as given a type of numeric expressions `exp` with typical member e, and a type of boolean expressions `bexp` with typical member b. We will further assume that these expressions contain all of the normal variables, constants, and operators.

These constructs are mostly standard. Informally, the **skip** command has no effect on the state. **abort** causes an immediate abnormal termination of the program. $x := e$ evaluates the numeric expression e and assigns the value to the variable x. $c_1 \; ; \; c_2$ executes command c_1 first, and if it terminates, then executes c_2. The conditional command **if** b **then** c_1 **else** c_2 **fi** first evaluates the boolean expression b; if it is true, then c_1 is executed, otherwise c_2 is executed. Finally, the iteration command **assert** a **while** b **do** c **od** evaluates b; if it is true, then the body c is executed, followed by executing the whole iteration command again, until b evaluates to false. The '**assert** a' phrase of the iteration command does not affect its execution; this is here as an annotation to aid the verification condition generator. The significance of a is to denote an *invariant*, a condition that is true every time control passes through the head of the loop.

Annotations are written in an *assertion language* that is a partner to this programming language. The assertion language is used to expresses conditions that are true at particular moments in a program's execution. Usually these conditions are attached to specific points in the control structure, signifying that whenever control passes through that point, then the attached assertion evaluates to true. For this simple example, we will take the assertion language to be the first-order predicate logic with operators for the normal numeric and boolean operations. In particular, $a_1 => a_2 \mid a_3$ is a conditional expression, which first evaluates a_1, and then yields the value of a_2 or a_3 depending on whether a_1 was true or false, respectively. We also specifically include the universal and existential quantifiers, ranging over nonnegative integers. We denote the types of numeric expressions and boolean expressions in the assertion language as `vexp` and `aexp`, respectively, with typical members v and a. We will also use p and q occasionally as typical members of `aexp`.

We use the same operator symbols (like "+") in the programming and assertion languages, overloading the operators and relying on the reader to disambiguate them by context.

2.2 Semantics

The execution of programs depends on the state of the computer's memory. In this simple programming language, all variables have nonnegative integer values. Following the notation of HOL, we will denote the type of nonnegative integers by `num` and the type of truth values as `bool`. We take the type of variables to be `var`, without specifying them completely at this time. Then we can represent states as functions of type $\texttt{state} = \texttt{var} \rightarrow \texttt{num}$, and we can refer to the value of a variable x in a state s as $s\ x$, using simple juxtaposition to indicate the application of a function to its argument.

Numeric and boolean expressions are evaluated by the curried functions E and B, respectively. Because these expressions may contain variables, their evaluation must refer to the current state.

$E\ e\ s = n$	Numeric expression e:`exp` evaluated in state s yields numeric value n:`num`.
$B\ b\ s = t$	Boolean expression b:`bexp` evaluated in state s yields truth value t:`bool`.

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

$$(f[e/x])(y) = \begin{cases} e & \text{if } y = x, \text{ and} \\ f(y) & \text{if } y \neq x \end{cases}$$

The operational semantics of the programming language is expressed by the following relation:

$C\ c\ s_1\ s_2$	Command c executed in state s_1 yields resulting state s_2.

Skip:

$$\frac{}{C\ \mathbf{skip}\ s\ s}$$

Abort:
(no rules)

Assignment:

$$\frac{}{C\ (x := e)\ s\ s[(E\ e\ s)/x]}$$

Sequence:

$$\frac{C\ c_1\ s_1\ s_2, \qquad C\ c_2\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ = \mathrm{T}, \qquad C\ c_1\ s_1\ s_2}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ s_1\ s_2}$$

$$\frac{B\ b\ s_1 = \mathrm{F}, \qquad C\ c_2\ s_1\ s_2}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ s_1\ s_2}$$

Iteration:

$$\frac{\begin{array}{c} B\ b\ s_1 = \mathrm{T}, \qquad C\ c\ s_1\ s_2 \\ C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ s_2\ s_3 \end{array}}{C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ s_1\ s_3}$$

$$\frac{B\ b\ s_1 = \mathrm{F}}{C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ s_1\ s_1}$$

Table 2.2: Example programming language structural operational semantics.

Table 2.2 gives the structural operational semantics of the programming language, specified by rules inductively defining the relation C.

The semantics of the assertion language is given by recursive functions V and A defined on the structure of `vexp` and `aexp`, in a directly denotational fashion. Since the expressions may contain variables, their evaluation must refer to the current state.

$V\ v\ s = n$	Numeric expression v:`vexp` evaluated in state s yields numeric value n:`num`.
$A\ a\ s = t$	Boolean expression a:`aexp` evaluated in state s yields truth value t:`bool`.

This syntax and structural operational semantics is the foundational definition for this programming language and its meaning. It is complete, in that we know the details of any prospective computation, given the initial state and the program to be executed. However, it is not the easiest form with which to reason about the correctness of programs. For that, we need to turn to a more abstract representation of the semantics, such as Hoare-style program logics.

2.3 Partial and Total Correctness

When talking about the correctness of a program, exactly what is meant? In general, this describes the consistency of a program with its specification. There have developed two versions of the specific meaning of correctness, known as *partial correctness* and *total correctness*. *Partial correctness* signifies that every time you run the program, every answer that it gives you is consistent with what is specified. However, partial correctness admits the possibility of not giving you any answer at all, by permitting the possibility of the program not terminating. A program that does not terminate is still said to be partially correct. In contrast, *total correctness* signifies that every time you start the program, it will in fact terminate, *and* the answer it gives you will be consistent with what is specified.

The partial and total correctness of commands may be expressed by logical formulae called *Hoare triples*, each containing a precondition, a command, and a postcondition. The precondition and postcondition are boolean expressions in the assertion language. Traditionally, the precondition and postcondition are written with curly braces ({ }) around them to signify partial correctness, and with square braces ([ ]) to signify total correctness. For our example programming language and its assertion language, we define notations for partial and total correctness

$\{a\}$	$=$	**close** $a \;=\; \forall s.\ A\ a\ s$
$\{p\}\ c\ \{q\}$	$=$	$\forall s_1\ s_2.\ A\ p\ s_1 \wedge C\ c\ s_1\ s_2 \Rightarrow A\ q\ s_2$
$[p]\ c\ [q]$	$=$	$(\forall s_1\ s_2.\ A\ p\ s_1 \wedge C\ c\ s_1\ s_2 \Rightarrow A\ q\ s_2)$
		$\wedge\ (\forall s_1.\ A\ p\ s_1 \Rightarrow (\exists s_2.\ C\ c\ s_1\ s_2))$

Table 2.3: Floyd/Hoare Partial and Total Correctness Semantics.

in Table 2.3.

As described in the table, we use $\{a\}$ to denote a boolean assertion expression which is true in all states. This is the same as having all of the free variables of a universally quantified, and so this is also known as the *universal closure* of a. **close** a denotes the same universal closure, but by means of a unary operator.

With these partial and total correctness notations, it now becomes possible to express an axiomatic semantics for a programming language, as a Hoare-style logic, which we will do in the next section.

In this dissertation, we will study a larger programming language that will include procedures with parameters. Verifying these procedures will introduce several new issues. It is an obvious but nevertheless significant feature that a procedure call has a semantics which depends on more than the syntactic components of the call itself—it must refer to the declaration of the procedure, which is external and part of the global context. This is unlike all of the constructs in the small example programming language given above.

The parameters to a procedure will include both value parameters, which are passed by value, and variable parameters, which are passed by name to simulate call-by-reference. The passing of these parameters, and their interaction with global variables, has historically been a delicate issue in properly defining Hoare-style rules for the semantics of procedure call. The inclusion of parameters also raises the need to verify that no aliasing has occurred between the actual variables presented in each call and the global variables which may be accessed from the body of the procedure, as aliasing greatly complicates the semantics in an intractable fashion.

To verify total correctness, it is necessary to prove that every command terminates, including procedure calls. If the termination of all other commands is established, a procedure call will terminate unless it initiates an infinitely descending sequence of procedure calls, which continue issuing new calls deeper and deeper and never finishing them. To prove termination, we must prove this infinite recursive descent does not occur. This will constitute a substantial portion of this dissertation's work, as we describe a new method for proving the termination of procedure calls which we believe to be simpler, more general, and easier to use than previous proposals.

2.4 Hoare Logics

In [Hoa69], Hoare presented a way to represent the calculations of a program by a series of manipulations of logical formulae, which were symbolic representations of sets of states. The logical formulae, known as "axioms" and "rules of inference," gave a simple and beautiful way to express and relate the sets of possible program states at different points within a program. In fact, under certain conditions it was possible to completely replace a denotational or operational definition of the semantics of a language with this "axiomatic" semantics. Instead

Skip: $$\frac{}{\{q\}\ \mathbf{skip}\ \{q\}}$$	*Conditional:* $$\frac{\{p \wedge b\}\ c_1\ \{q\} \quad \{p \wedge \sim b\}\ c_2\ \{q\}}{\{p\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ \{q\}}$$
Abort: $$\frac{}{\{\mathbf{false}\}\ \mathbf{abort}\ \{q\}}$$	*Iteration:* $$\frac{\{a \wedge b\}\ c\ \{a\} \quad \{a \wedge \sim b \Rightarrow q\}}{\{a\}\ \mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}\ \{q\}}$$
Assignment: $$\frac{}{\{q \lhd [e/x]\}\ x := e\ \{q\}}$$	*Precondition Strengthening:* $$\frac{\{p \Rightarrow a\} \quad \{a\}\ c\ \{q\}}{\{p\}\ c\ \{q\}}$$
Sequence: $$\frac{\{p\}\ c_1\ \{r\}, \quad \{r\}\ c_2\ \{q\}}{\{p\}\ c_1\ ;\ c_2\ \{r\}}$$	

Table 2.4: Example programming language axiomatic semantics.

of involving states, these "rules" now dealt with symbolic formulae representing sets of possible states. This had the benefit of more closely paralleling the reasoning needed to actually prove a program correct, without being as concerned with the details of actual operational semantics. To some, reasoning about states seemed "lower level" and more representation-dependent than reasoning about expressions denoting relationships among variables.

To illustrate these ideas, consider the Hoare logic in Table 2.4 for the simple programming language we have developed so far.

In the rule for Assignment, the precondition is $q \lhd [e/x]$. $\lhd$ denotes the operation of proper substitution; hence, this denotes the proper substitution of the expression e for the variable x throughout the assertion q. There is one small problem with this, which is that the expressions e and q are really from two different, though related, languages. We will intentionally gloss over this issue now, simply using e as a member of both languages. This also applies to b where it appears in the Conditional and Iteration rules.

Given these rules, we may now compose them to prove theorems about structured commands. For example, from the Rule of Assignment, we have

$$\{x0 = 0 * y0 + x \wedge y0 = y\}\ r := x\ \{x0 = 0 * y0 + r \wedge y0 = y\}$$

and

$$\{x0 = 0 * y0 + r \wedge y0 = y\}\ q := 0\ \{x0 = q * y0 + r \wedge y0 = y\}.$$

From these and the Rule of Sequence, we have

$$\{x0 = 0 * y0 + x \wedge y0 = y\}\ r := x\ ;\ q := 0\ \{x0 = q * y0 + r \wedge y0 = y\}.$$

For completeness, a Hoare logic will usually contain additional rules not based on particular commands, such as precondition strengthening or postcondition weakening. The Precondition Strengthening Rule in Table 2.4 is an example.

2.5 Soundness and Completeness

An axiomatic semantics for a programming language has the benefit of better supporting proofs of program correctness, without involving the detailed and seemingly mechanical apparatus of operational semantics. However, with this benefit of abstraction comes a corresponding weakness. The very fact that the new Hoare rules are more distant from the operational details means a greater possibility that in fact they might not be logically consistent. This question of consistency has two aspects, which are called *soundness* and *completeness*. *Soundness* is the quality that every rule in the axiomatic semantics is true for every possible computation described by the foundational operational semantics. A rule is sound if every computation that satisfies the antecedents of the rule also satisfies its consequent. *Completeness* is the quality of the axiomatic semantics of being expressive and powerful enough to be able to prove within the Hoare logic theorems that represent every computation allowed by the operational semantics. One could easily come up with a sound axiomatic semantics by having only a few trivial rules; but then one would never be able to derive useful results about interesting programs. Likewise, one could come up with powerful axiomatic semantics with which many theorems about programs could be proven; but if any one rule is not sound, the entire system is useless.

Of these two qualities, we have chosen for this dissertation to concentrate on soundness. By this choice, we do not intend to minimize the role or importance of completeness—it is simply a question of not being able to solve every problem at once. Nevertheless, we do feel that of the two qualities, soundness is in some sense the more vital one. A system that is sound but not complete may still be useful for proving many programs correct. A system that is complete but not sound will give you the ability to prove many seemingly powerful theorems about programs which are in fact not true with respect to the operational semantics.

Also, researchers have occasionally proposed rules for axiomatic semantics which were later found to be unsound. This problem has arisen, for example, in describing the passing of parameters in procedure calls. This history shows a need for some mechanism to more carefully establish the soundness of the rules of an axiomatic semantics, thereby establishing the rules as trustworthy, since all further proof efforts in that language depend on them.

2.6 Verification Condition Generators

Given a Hoare logic for a particular programming language, it may be possible to partially automate the process of applying the rules of the logic to prove the correctness of a program. Generally this process is guided by the structure of the program, applying in each case the Hoare logic rule for the command which is the major structure of the phrase under consideration.

A Verification Condition Generator takes a suitably annotated program and its specification, and traces a proof of its correctness, according to the rules of the language's axiomatic semantics. Each command has its own appropriate rule which is applied when that command is the major structure of the current proof goal. This replaces the current goal by the antecedents of the Hoare rule. These antecedents then become the subgoals to be resolved by further applications of the rules of the logic.

At certain points, the rules require that additional conditions be met; for example, in the Iteration Rule in Table 2.4, there is the antecedent $a \wedge \sim b \Rightarrow q$. This is not a partial correctness formula, and so cannot be reduced further by rules of the Hoare logic. The VCG emits this as a verification condition to be proven by the user.

As an example, we present in Table 2.5 a Verification Condition Generator for the simple programming language discussed so far. It consists of two functions,

$vcg1$	$vcg1\ (\mathbf{skip})\ q = q,\ [\,]$ $vcg1\ (\mathbf{abort})\ q = \mathbf{true},\ [\,]$ $vcg1\ (x := e)\ q = q \lhd [e/x],\ [\,]$ $vcg1\ (c_1\,;\,c_2)\ q = \mathbf{let}\ (r, h_2) = vcg1\ c_2\ q\ \mathbf{in}$ $\qquad \mathbf{let}\ (p, h_1) = vcg1\ c_1\ r\ \mathbf{in}$ $\qquad p,\ (h_1\ \&\ h_2)$ $vcg1\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ q =$ $\qquad \mathbf{let}\ (r_1, h_1) = vcg1\ c_1\ q\ \mathbf{in}$ $\qquad \mathbf{let}\ (r_2, h_2) = vcg1\ c_2\ q\ \mathbf{in}$ $\qquad (b\ \ {=}{>}\ \ r_1\ \mid\ r_2),\ (h_1\ \&\ h_2)$ $vcg1\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ q =$ $\qquad \mathbf{let}\ (p, h) = vcg1\ c\ a\ \mathbf{in}$ $\qquad a,\ [a \wedge b \Rightarrow p\,;$ $\qquad\quad a\ \wedge \sim b \Rightarrow q]\ \&\ h$
vcg	$vcg\ p\ c\ q = \mathbf{let}\ (r, h) = vcg1\ c\ q\ \mathbf{in}\ [p \Rightarrow r]\ \&\ h$

Table 2.5: Example Verification Condition Generator.

the main function vcg and a helper function $vcg1$. The square brackets [and] enclose a list, for which semicolons (';') separate list elements; the phrase [] denotes an empty list. Comma (',') creates a pair, and ampersand ('&') appends two lists together.

$vcg1$ has type `cmd`$\rightarrow$`aexp`$\rightarrow$(`aexp` $\times$ `(aexp)list`). This function takes a command and a postcondition, 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 and postcondition. This function does most of the work of calculating verification conditions.

$vcg1$ is called by the main verification condition generator function, vcg, with type `aexp`$\rightarrow$`cmd`$\rightarrow$`aexp`$\rightarrow$`(aexp)list`. vcg takes a precondition, a command, and a postcondition, and returns a list of the verification conditions for that command.

Given such a Verification Condition Generator, there are two interesting things we might ask about it. First, does the truth of the verification conditions it generates in fact imply the correctness of the program? If so, then we say the VCG is *sound*. Second, if the program is in fact correct, does the VCG generate verification conditions sufficient to prove the program correct from the axiomatic semantics? We call such a VCG *complete*. In this dissertation, we will only focus on the first question, that of soundness.

2.7 Higher Order Logic

Higher Order Logic (HOL) is a mechanical proof assistant that mechanizes higher order logic, and provides an environment for defining systems and proving statements about them. It is secure in that only true theorems may be proven, and this security is ensured at each point that a theorem is constructed.

HOL has been applied in many areas. The first and still most prevalent use is in the area of hardware verification, where it has been used to verify the correctness of several microprocessors. In the area of software, HOL has been applied to Lamport's Temporal Logic of Actions (TLA), Chandy and Misra's UNITY language, Hoare's CSP, and Milner's CCS and π-calculus. HOL is one of the oldest and most mature mechanical proof assistants available, roughly comparable in maturity and degree of use with the Boyer-Moore Theorem Prover [BM88]. Many other proof assistants have been introduced more recently that in some ways surpass HOL, but HOL has one of the largest user communities and history of experience. We therefore considered it ideal for this work.

HOL differs from the Boyer-Moore Theorem Prover in that HOL does not attempt to automatically prove theorems, but rather provides an environment and supporting tools to the user to enable him to prove the theorems. Thus, HOL is better described as a mechanical proof assistant, recording the proof efforts and its products along the way, and maintaining the security of the system at each point, but remaining essentially passive and directed by the user. It is, however, powerfully programmable, and thus the user is free to construct programs which automate whatever theorem-proving strategy he desires.

2.7.1 Higher Order Logic as a Logic

Higher Order Logic is a version of predicate calculus which allows quantification over predicate and function symbols of any order. It is therefore an *ω-order* *logic,* or *finite type theory,* according to Andrews [And86]. In such a type theory, all variables are given types, and quantification is over the values of a type. Type theory differs from set theory in that functions, not sets, are taken as the most elementary objects. Some researchers have commented that type theory seems to more closely and naturally parallel the computations of a program than set theory. A formulation of type theory was presented by Church in [Chu40]. Andrews presents a modern version in [And86] which he names $\mathcal{Q}_0$. The logic implemented in the Higher Order Logic system is very close to Andrews' $\mathcal{Q}_0$. This logic has the power of classical logic, with an intuitionistic style. The logic has the ability to be extended by several means, including the definition of new types and type constructors, the definition of new constants (including new functions and predicates), and even the assertion of new axioms.

The HOL logic is based on eight rules of inference and five axioms. These are the core of the logical system. Each rule is sound, so one can only derive true results from applying them to true theorems. As the HOL system is built up, each new inference rule consists of calls to previously defined inference rules, ultimately devolving to sequences of these eight primitive inference rules. Therefore the HOL proof system is fundamentally sound, in that only true results can be proven.

HOL provides the ability to assert new axioms; this is done at the user's discretion, and he then bears any responsibility for possible inconsistencies which may be introduced. Since such inconsistencies may be hard to predict intuitively, we have chosen in our use of the HOL system to restrict ourselves to never using the ability to assert new axioms. This style of using HOL is called a "definitional" or "conservative extension," because it is assured of never introducing any inconsistencies. In a conservative extension, the security of HOL is not compromised, and hence the basic soundness of HOL is maintained.

We will not describe in detail the theoretical foundation of the HOL logic, referring the interested reader to [GM93], because the purpose of this dissertation is not the study of HOL itself, but rather its application as a tool to support the verification of VCGs. Hence we will concentrate on describing the useful aspects of HOL that apply to our work.

2.7.2 Higher Order Logic as a Mechanical Proof Assistant

The HOL system provides the user a logic that can easily be extended, by the definition of new functions, relations, and types. These extensions are organized into units called *theories*. Each theory is similar to a traditional theory of logic, in that it contains definitions of new types and constants, and theorems which follow from the definitions. It differs from a traditional theory in that a traditional theory is considered to contain the infinite set of all possible theorems which could be proven from the definitions, whereas a theory in HOL contains only the subset which have been actually proven using the given rules of inference and other tools of the HOL system.

When the HOL system it started, it presents to the user an interactive programming environment using the programming language ML, or *Meta Language* of HOL. The user types expressions in ML, which are then executed by the system, performing any side effects and printing the value yielded. The language ML contains the data types `term` and `thm`, which represent terms and theorems in the HOL logic. These terms represent a second language, called the *Object Language*

(OL) of HOL, embedded within ML. ML functions are provided to construct and deconstruct terms of the OL language. Theorems, however, may not be so freely manipulated. Of central importance is the fact that theorems, objects of type `thm`, can only be constructed by means of the eight standard rules of inference. Each rule is represented as a ML function. Thus the security of HOL is maintained by implementing `thm` as an abstract data type.

Additional rules, called *derived rules of inference*, can be written as new ML functions. A derived rule of inference could involve thousands of individual calls to the eight standard rules of inference. Each rule typically takes a number of theorems as arguments and produces a theorem as a result. This methodology of producing new theorems by calling functions is called *forward proof*.

One of the strengths of HOL is that in addition to supporting forward proof, it also supports *backwards proof*, where one establishes a goal to be proved, and then breaks that goal into a number of subgoals, each of which is refined further, until every subgoal is resolved, at which point the original goal is established as a theorem. At each refinement step, the operation that is applied is called in HOL a *tactic*, which is a function of a particular type. The effect of applying a tactic is to replace a current goal with a set of subgoals which if proven are sufficient to prove the original goal. The effect of a tactic is essentially the inversion of an inference rule. Tactics may be composed by functions called *tacticals*, allowing a complex tactic to be built to prove a particular theorem.

Functions in ML are provided to create new types, make new definitions, prove new theorems, and store the results into theories on disk. These may then be used to support further extensions. In this incremental way a large system may be constructed.

2.8 Embeddings

Previous researchers have constructed representations of programming languages within HOL, of which the work of Gordon [Gor89] was seminal. He introduced new constants in the HOL logic to represent each program construct, defining them as functions directly denoting the construct's semantic meaning. This is known as a "shallow" embedding of the programming language in the HOL logic, using the terminology described in [BGG+92]. This approach yielded tools which could be used to soundly verify individual programs. However, there were certain fundamental limitations to the expressiveness of this approach, and to the theorems which could be proven about all programs. This was recognized by Gordon himself [Gor89]:

> $\mathcal{P}[\mathcal{E}/\mathcal{V}]$ (substitution) is a meta notation and consequently the assignment axiom can only be stated as a meta theorem. This elementary point is nevertheless quite subtle. In order to prove the assignment axiom as a theorem within higher order logic it would be necessary to have types in the logic corresponding to formulae, variables and terms. One could then prove something like:
>
> $\vdash \forall P\ E\ V.\, \mathsf{Spec}\,(\mathsf{Truth}\,(\mathsf{Subst}\,(P, E, V)),\ \mathsf{Assign}\,(V,\ \mathsf{Value}\ E),\ \mathsf{Truth}\ P)$
>
> It is clear that working out the details of this would be a lot of work.

This dissertation explores the alternative approach described but not investigated by Gordon. It yields great expressiveness and control in stating and proving as theorems within HOL concepts which previously were only describable as meta-theorems outside HOL. For example, we have proven the assignment axiom described above:

$$\vdash \forall q\ x\ e.\ \{q \lhd [x := e]\}\ x := e\ \{q\}$$

where $q \lhd [x := e]$ is a substituted version of q, described later.

To achieve this expressiveness, it is necessary to create a deeper foundation than that used previously. Instead of using an extension of the HOL Object Language as the programming language, we create an entirely new set of datatypes within the Object Language to represent constructs of the programming language and the associated assertion language. This is known as a "deep" embedding, as opposed to the shallow embedding developed by Gordon. This allows a significant difference in the way that the semantics of the programming language is defined. Instead of defining a construct *as* its semantics meaning, we define the construct as simply a syntactic constructor of phrases in the programming language, and then separately define the semantics of each construct in a structural operational semantics. This separation means that we can now decompose and analyze syntactic program phrases at the HOL Object Language level, and thus reason within HOL about the semantics of purely syntactic manipulations, such as substitution or verification condition generation, since they exist *within* the HOL logic.

This has definite advantages because syntactic manipulations, when semantically correct, are simpler and easier to calculate. They encapsulate a level of detailed semantic reasoning that then only needs to be proven once, instead of having to be repeatedly proven for every occurrence of that manipulation. This will be a recurring pattern in this dissertation, where repeatedly a syntactic manipulation is defined, and then its semantics is described, and proved correct within HOL.

CHAPTER 3

Survey of Previous Research

> "Now if anyone builds on this foundation with gold, silver, precious stones, wood, hay, straw, each one's work will become clear; for the Day will declare it, because it will be revealed by fire; and the fire will test each one's work, of what sort it is."
>
> — 1 Corinthians 3:12–13

In this chapter we discuss the work that has been done by others that supports proofs of program correctness for programs containing various language features. The research discussed below include expressions with side effects, procedures with variable and value parameters, including especially the total correctness of mutually recursive procedures, verification condition generators, embeddings, and mechanically verified axiomatic semantics. These areas have been developed in varying degrees, from fairly deep descriptions of the partial correctness of procedures, to an apparent lack of development of expressions with side effects. In all these areas we hope to give a perspective on the context in which our research was conducted.

3.1 Expressions with Side Effects

Expressions have typically not been treated as a highlight in previous work on verification; there are some exceptions, notably Sokołowski [Sok84]. Even he does not treat expressions with side effects. Side effects appear commonly in actual programming languages, such as C or C++, with the operators ++ and get_ch. In addition, several interesting functions are naturally designed with a side effect; an example is the standard method for calculating random numbers, based on a seed which is updated each time the random number generator is run.

In general, expressions with side effects have been explicitly excluded, from the original paper by Hoare [Hoa69], through Dijkstra's work [Dij76], and continuing through that of Alagić and Arbib [AA78], de Bakker [dB80], Gries [Gri81], Gordon [Gor88], Apt and Olderog [AO91], and Dahl [Dah92].

Since expressions did not have side effects, they were often considered to be a sublanguage, common to both the programming language and the assertion language. Thus one would commonly see expressions such as $p \wedge b$, where p was an assertion and b was a boolean expression from the programming language.

One of the key realizations of this work was the need to carefully distinguish these two languages, and not confuse their expression sublanguages. This then requires us to *translate* programming language expressions into the assertion language before the two may be combined as above. This is described in detail in Section 10.3 on Translations.

3.2 Procedures

The treatment of procedures by different authors has varied in the aspects addressed and in their depth. Some have dealt with parameters, some have not. Some methods handle recursive procedures, but not mutual recursion, and others do. Some treatments have been explicitly detailed, including such complexities as the subtleties of proper substitution and the generation of new variable names; other discussions have concentrated on providing a more intuitive, high-level view of the proof process. Partial correctness has been generally well analyzed, but termination has been treated by relatively few authors.

Hoare's original paper [Hoa69] did not cover procedures, but with foresight described how the correct specification and proof of the correctness of procedures could be an essential building block in the proof of large programs, as well as providing aid in documentation and in code modification. Hoare saw that the structure of the proof would mirror the structure of the procedures. In [Hoa71], he gave an axiomatic approach to recursive procedures, and this has been the style generally used since.

Current versions of Hoare's rules for the partial correctness of procedures including parameters are presented by Francez [Fra92]. For illustration, and leaving out several details, we have adapted his rules into our notation as follows. We take the syntax of a procedure call to be **call** *proc* $(x; e)$, where x is a list of variables and e is a list of expressions. *proc* is the name of a procedure defined as **procedure** *proc* $(y; z); c$, where y is a list of the *variable formal parameters*, using call by value-result to pass the parameters, z is a list of the *value formal parameters*, using call by value, and c is the body of the procedure, a command.

Rule of Recursion:

$$\frac{\{pre\}\ \mathbf{call}\ proc\ (y;z)\ \{post\} \vdash \{pre\}\ c\ \{post\}}{\{pre\}\ \mathbf{call}\ proc\ (y;z)\ \{post\}}$$

provided that the program contains the declaration **procedure** $proc\ (y;z); c$, and $FV(pre) \subseteq y \cup z$, $FV(post) \subseteq y$. This is actually a *meta-rule*, which has a "provability" claim as one of its assumptions. This provability claim is a side proof, where one may use $\{pre\}$ **call** $proc\ (y;z)\ \{post\}$ as an assumption in proving $\{pre\}\ c\ \{post\}$. This rule is the verification of the partial correctness of the body of $proc$, c, with respect to precondition pre and postcondition $post$. This rule is then adapted to particular calls by the following rule:

Rule of Adaptation:

$$\frac{\{pre\}\ \mathbf{call}\ proc\ (y;z)\ \{post\}}{\{(pre \lhd [e/z]) \wedge (\forall a.\ (post \lhd [a/y]) \Rightarrow (q \lhd [a/x]))\}\ \mathbf{call}\ proc\ (x;e)\ \{q\}}$$

with additional restrictions, such as non-aliasing.

This approach to proving the correctness of procedures has been generally adopted, and every other treatment we have studied used some variation of these rules. However, Sokołowski has remarked [Sok77] that it is not clear what meaning is assigned to $\{pre\}$ **call** $proc\ (y;z)\ \{post\} \vdash \{pre\}\ c\ \{post\}$ that appears in the Rule of Recursion. Francez [Fra92] explains the Rule of Recursion as a meta-rule, one of whose antecedents is not merely a correctness assertion, but instead is a statement about the existence of a proof from an assumption, namely that if one assumes the partial correctness specification about the invocation command, then the same specification is provable about the body of the invoked procedure. This is handled as a separate or side proof, which must be completed before making the application of the Rule of Recursion.

We found this approach to be difficult to break down into a standard method of solution, and therefore not easily adaptable for a VCG. Instead, we handle the problem of the order of proof of the body versus the call by a meta-level proof done once to verify the VCG.

The treatment of procedures has historically been fraught with unsoundness, as noted by Francez [Fra92]:

> Another indication of the intricacy of rules dealing with the language constructs considered in this chapter [on procedures] is that several wrong rules have been proposed, the errors in which were caught much later. However, any serious methodological attempt at verification of actual software will have to deal with such mechanisms to be of any practical use. Thus, awareness of complications and limitations is of crucial importance when programs with procedures are concerned.

We believe that this history of unsoundness from capable researchers is a strong indication of an inherent underlying degree of complexity which requires powerful tools. The treatment of procedures is an area where the security of a mechanical proof-checker has been of great value to us.

3.3 Total Correctness of Mutually Recursive Procedures

Proving the total correctness of mutually recursive procedures involves showing that they terminate, in addition to their partial correctness. Mutually recursive procedures may not terminate if a computation follows a cycle of procedures in the procedure call graph, where the procedures repeatedly call each other in that cycle without ever returning. We call this situation *infinite recursive descent*.

The general strategy to prevent this infinite recursive descent is to limit the possible depth of calls that such a calling chain can descend. Any finite limit is sufficient to guarantee termination. A powerful and general technique to impose such a limit is to track the procedures in the calling chain, attaching a value to each procedure, where the values are all taken from a well-founded set, and where the values strictly decrease along the chain. By the definition of a well-founded set, there do not exist any infinite descending sequences of values from the well-founded set, and so the situation of such an infinite chain of procedure calls can not occur.

To specify this, one chooses an expression whose value is in the well-founded set, and considers the value attached to each procedure to be the value of the expression at the head of the procedure, when it is entered. In the past, most reseachers have limited the choice of well-founded set to be the nonnegative integers. In addition, most researchers have chosen the ordering relation of the well-founded set to be the successor relation, where the only pairs in the relation were of the form $(n,\ n+1)$ for $n \geq 0$. These are useful choices for exploration, but they can also occlude the fact that there is a great deal more power available in the more general well-founded set.

3.3.1 Sokołowski

For termination, the original work was done by Sokołowski [Sok77], where he introduced a recursion depth counter. This depth counter was a measure of how much more deeply the computation could issue calls. For each call, the depth counter was decreased by one, with the invariant maintained that it remained nonnegative. Since any number cannot be decreased indefinitely without becoming negative (an example of a well-foundedness argument), the procedure could be proven to terminate. Sokołowski gave a rule of procedure recursion that supported a termination argument. His rule was based on Hoare's, and had the following form, adapted to the style used above.

$$\frac{\begin{array}{c}\{pre(0)\}\ c\ \{post\} \\ \{pre(i)\}\ \mathbf{call}\ proc\ \{post\} \vdash \{pre(i+1)\}\ c\ \{post\}\end{array}}{\{\exists i \geq 0.\ pre(i)\}\ \mathbf{call}\ proc\ \{post\}}$$

The recursion depth counter is represented by the argument to the precondition *pre*. Sokołowski then extended this rule to systems of mutually recursive procedures by reinterpreting the elements of the rule as vectors. He gave proofs of soundness and completeness of the new rule.

Sokołowski spent some time discussing the fact that the provability claim in the above rule did not concern programs, but the inference system for reasoning about programs. He resolved this trouble by describing an infinite sequence of predicate transformers, and modified the rule to depend on all the predicate transformers.

This system did not deal with parameters.

3.3.2 Apt

In 1981, Apt [Apt81] proved that Sokołowski's rule did not have sufficient strength to be able to prove all valid correctness specifications, i.e., that it was not complete. Apt then added additional proof rules, still not including parameters, to deal with the effects of procedure calls on variables not used in the procedure.

3.3.3 America and de Boer

In 1990, America and de Boer [AdB90] noted that the augmented system presented by Apt was not sound, that one could derive from it correctness specifications which were not valid. An example of such a derivation was described in their work. They then presented a modification of Apt's proof system with some restrictions added, and proved the resulting system was both sound and complete. This paper was quite comprehensive and thorough in its treatment. However, its scope was limited in several ways; the set of declared procedures was restricted to a single procedure, parameters were not addressed, and continuing the tradition set by Sokołowski, the recursion depth counter was required to decrease by exactly one for every individual procedure call.

3.3.4 Pandya and Joseph

During this discussion of soundness and completeness, Pandya and Joseph [PJ86] considered a new aspect of the problem of proving the total correctness of recursive procedures, namely the simplicity and ease of applying the proof techniques. They found that even for simple programs, that Sokołowski's rule could require the use of complex predicates to encode information about the depth counter, to ensure that it decreased by exactly one for each procedure call. This significantly added to the difficulty of practically proving such programs. Pandya and Joseph noted that this requirement of decreasing by one did not consider the structure of the program itself, and thus was *data-directed* as opposed to being *syntax-directed*. They proposed a new rule, based on choosing a subset of the procedures called the *header* procedures. Every cycle in the procedure call graph was required to contain at least one header procedure. Then the requirement of decreasing by one was applied to only the header procedures, and not the rest. This enabled much simpler descriptions of the recursion depth counter, making proofs more natural. Pandya and Joseph's approach did require the programmer to select a valid set of header procedures for a program, but they described algorithms to help identify such a set. Still, this was an additional burden on the programmer, and varied in its effectiveness based on the particular structure of the program being proved. In the worst case, one would need to choose all procedures as being header procedures, in which case their rule simplified to Sokołowski's.

3.4 Verification Condition Generators, Embeddings, and Mechanically Verified Axiomatic Semantics

Verification Condition Generators have a long and respectable history. They first appeared in the early 1970's, of which Igarashi, London, and Luckham's VCG [ILL75] is a notable and characteristic example. In the beginning they were hailed as an answer to the difficulty of proving programs correct. This hope waned over time, however. First of all, it was discovered that for many simple programming languages, the work done by the VCG was mostly trivial and not hard to do by hand. Then, even after the VCG had done its work and reduced the problem of proving the program to the problem of proving the verification conditions, that those verification conditions were not always easy to prove, and could contain the bulk of the necessary effort of the entire proof. An additional feature that was not discussed as much was the fact that for the most part, these verification condition generators were not themselves verified. This meant that any proof using and relying on these VCG tools might not be sound, even if all the verification conditions were correctly proven. Ragland's work [Rag73] in 1973 is a notable exception to this, far ahead of its time.

Finally, a verification condition generator is usually based on an axiomatic semantics for the programming language. When these programming languages were extended to include procedure calls (an obvious necessity), a disturbing number of the rules proposed for procedure calls turned out to be unsound. It became evident that the area of procedure calls was more complicated than had originally appeared. Given these difficulties, interest declined in the use of VCGs, and research mostly turned to other subjects, such as discovering rules to handle concurrency in various forms.

In recent years, there have been several shallow embeddings of programming languages in the HOL theorem proving environment, including the creation of verification condition generators. These have taken the form of HOL tactics, which in general reduce a current goal to be proved to a sufficient set of subgoals. In contrast to the traditional VCGs created as stand-alone programs, these HOL VCGs had their soundness secured by the inherent security of the HOL system itself. This was a very significant advantage. No verification of the VCG itself was necessary, as every application of the tactic would prove all necessary subsidiary theorems as part of the process. However, this also was a weakness of the HOL VCGs, because it required that every proof be carried out at the semantic level, instead of the syntactic manipulations that were simpler and that were the traditional work of VCGs. Also, these semantic VCGs required an additional degree of annotation and specification from the user beyond what had been required by the syntactic VCGs.

In addition, there have been forays into the areas of deep embeddings within HOL and into mechanical verification of axiomatic semantics, including concurrency, proven from the underlying operational semantics. These technologies have not usually been combined together with VCGs, however, and generally the verification of VCGs has not been targeted recently, until our work.

3.4.1 Ragland

Ragland in 1973 verified a verification condition generator [Rag73]. It was written in Nucleus, a language Ragland had invented to have the expressiveness to write a VCG, and also be verifiable itself. This was a remarkable piece of work, well ahead of its time. The VCG system consisted of 203 procedures, nearly all of which were less than one page long. These gave rise to approximately 4000 verification conditions. The proof of the VCG used an unverified VCG written in Snobol4. The 4000 verification conditions it generated were proven by Ragland by hand, not mechanically. In our opinion, this proof was a *tour de force*. This proof substantially increased the degree of trustworthiness of Ragland's VCG.

3.4.2 Igarashi, London, and Luckham

In 1975, Igarashi, London, and Luckham [ILL75] gave an axiomatic semantics for a substantial subset of Pascal which included procedures, and described a VCG for partial correctness that they had written in MLISP2. The soundness of the axiomatic semantics was verified by hand proof, but the correctness of the VCG was not rigorously proven. The only mechanized part of this work was the VCG itself. This paper has become a classic reference on VCGs.

3.4.3 Boyer and Moore

In 1981, Boyer and Moore presented a verification condition generator for a subset of ANSI FORTRAN 66 and 77 [BM81]. This produced verification conditions as goals for the Boyer-Moore theorem prover. The VCG was remarkable for several reasons, including the substantial coverage of much of a "real" programming language, the inclusion of a static check of the syntax to enforce a set of syntactic restrictions (similar to our "well-formedness" constraints), the thorough analysis of aliasing, and the generation of verification conditions to prove termination. The approach to proving termination involved attaching "clocks" to various statements, which were expressions yielding values in a well-founded set, with the provision that every time control passed a clock, strictly less time was left on the clock than on the previous clock encountered.

This was a substantial and powerful VCG, with the advantage that the verification conditions generated could then be proven with the aid of the Boyer-Moore theorem prover. However, there was no formal axiomatic semantics presented to justify the operation of the VCG, and no verification of the VCG was considered.

3.4.4 Gray

In 1987, Gray presented a verification condition generator he had created [Gra87] to help teach axiomatic semantics to undergraduate students. The language considered resembled a subset of Pascal, and contained input and output commands, as well as procedure calls with both value and variable parameters. This stand-alone VCG was implemented by Gray and provided to his students. He wrote that

> In a teaching situation this VCG allows the students to concentrate on the tasks of specifying programs and proving lemmas, and relieves them of the tedious symbol manipulation required to generate the lemmas.

The verification conditions produced were to be proven by hand by the students. The issue of the verification of the VCG itself was not addressed by Gray, because it was not central to his goal of undergraduate education.

3.4.5 Gordon

Gordon in 1989 [Gor89] did the original work of constructing within HOL a framework for proving the correctness of programs. This was a seminal work, although it did not cover procedures. Gordon created a shallow embedding of the programming language considered, introducing new constants in the HOL logic to represent each program construct, defining them as functions directly denoting the construct's semantic meaning. This work included defining verification condition generators for both partial and total correctness as tactics. This approach yielded tools which could be used to soundly verify individual programs. However, the VCG tactic he defined was not itself proven. Its soundness was ensured by the security of HOL itself. The chief strength of this work was the ability to contain the entire proof of a program, from the original program correctness goal to the proof of the individual verification conditions, within a single mechanical proof checker.

3.4.6 Agerholm

In 1991 Agerholm [Age91] used a similar shallow embedding to define the weakest preconditions of a small **while**-loop language, including unbounded nondeterminism and blocks. The semantics was designed to avoid syntactic notions like substitution. Similar to Gordon's work, Agerholm defined a verification condition generator for total correctness specifications as an HOL tactic. This tactic needed the user to supply additional information to handle sequences of commands and the **while** command.

3.4.7 Melham

In 1992, Melham [Mel92] created a *deep* embedding of the π-calculus in HOL supporting meta-theoretic reasoning about the π-calculus itself. Melham was careful to explicitly define all syntactic operations within the logic, including substitution, which previous authors had avoided. He used simultaneous substitutions, and noted that this was one of the more complex definitions, due to the need to change bound names. There were several points where the work was automated, but no VCG of the traditional style was presented.

3.4.8 Camilleri and Melham

Also in 1992, Camilleri and Melham [CM92] created a library for HOL which supported the definition and use of relations inductively defined by rules. In this work, one of the examples presented was the definition of a structural operational semantics for a small language. The command structure was based on a deep embedding, although the authors did not use this term. From this definition, the authors proved the soundness of a Floyd-Hoare partial correctness rule for the **while** command.

3.4.9 Zhang, Shaw, Olsson, Levitt, *et. al.*

In 1993, Zhang, Shaw, Olsson, Levitt, Archer, Heckman, and Benson [ZSO+93] described a shallow embedding within HOL of the concurrent programming language microSR, a derivative of SR. This language used a message-passing mechanism, with asynchronous send and synchronous receive statements. Concurrent parts of the program could only communicate through this message-passing mechanism, with no shared globals. The Hoare logic for microSR was formally proven to be sound within HOL, a valuable achievement in the subtle area of concurrency. The work did not include a verification condition generator. The chief contribution of this paper was the substantial and important mechanical verification of the Hoare logic rules concerning concurrency.

3.4.10 Lin

Also in 1993, Lin [Lin93] presented a verification tool called VPAM for value-passing CCS, Milner's Calculus of Communicating Systems. This tool appears similar to verification condition generators. This was described in a paper by Nesi [Nes93] as follows:

> The verification tool VPAM for value-passing CCS is based on a proof system which deals with data and boolean expressions *symbolically*. This means that, when value-passing agents are analyzed, boolean

and value expressions are not evaluated, and input variables are not instantiated. In this way, reasoning about data is separated from reasoning about agents, and is performed by extracting "proof obligations" which can be verified by another theorem prover later or on-line with the main proof about the process behavior.

3.4.11 Kaufmann

In 1994, Kaufmann used the Boyer-Moore Theorem Prover to produce a VCG similar in style and concept to the VCGs produced for shallow embeddings in HOL [Kau94]. In this work, Kaufmann created a proof which was essentially a proof at the semantic level, but it was guided and aided automatically by the structure of the program by the VCG. The VCG acted as a heuristic guide to form the proof, so the security of the proof rested not on the unverified VCG, but on the security of the Boyer-Moore Theorem Prover.

3.4.12 Homeier and Martin

In 1994, we presented an early version of some of the work of this dissertation [HM94], for a standard **while**-loop programming language without procedures but containing expressions with side-effects. The rules of the Hoare logic presented were proven sound within HOL from an underlying structural operational semantics. As opposed to much previous work in HOL this was based on a *deep* embedding of the programming language in the HOL logic. The verified Hoare logic then formed an axiomatic semantics for partial correctness which supported the definition and proof of correctness for a verification condition generator function *within* the HOL logic. The theorem of the verification of the VCG stated that for any program and its specification, if all of the verification conditions the VCG generated were true, then the program was partially correct with respect to its specification. This theorem then supported the application of the VCG function to prove individual programs correct.

CHAPTER 4

Organization of Dissertation

"Let all things be done decently and in order."

— 1 Corinthians 14:40

The dissertation is organized as follows.

Part I describes the background of this work, including the technologies that underlie it and the previous research in this area. Part II presents our primary results, including the five program logics, the definition and proof of the VCG, and examples of its use. Part III is a tour of interesting aspects of the system; these are divided into those relevant to partial correctness and those supporting total correctness. Finally, Part IV presents our conclusions and possibilities for future research.

In Part I, Chapter 1 introduces and motivates the need for program correctness, and introduces the concept of verification condition generators. Chapter 2 describes the foundational technologies that underlie this work, such as structural operational semantics and Floyd/Hoare-style rules. Chapter 3 is a survey of previous research on verification condition generators, and in particular, methods to prove the total correctness of procedures. Chapter 4 gives the overall organization of the dissertation.

In Part II, Chapter 5 defines the syntax and semantics of the Sunrise programming language and assertion language. Chapter 6 presents the five program logics, with their fourteen correctness specifications, that can be used to prove Sunrise programs totally correct. Chapter 7 is the heart of this work. It defines a verification condition generator for the Sunrise system, and also presents theorems that verify it. Chapter 8 then takes this VCG and applies it to several examples, with transcripts. Chapter 9 describes where the source code of the Sunrise system may be found, for readers who may wish to use the system themselves to prove programs.

In Part III, Chapter 10 describes various aspects of the system relating to proving partial correctness. Chapter 11 then describes the proof of termination, presenting its essence.

In Part IV, Chapter 12 describes our sense of this work's significance. Chapter 13 examines the question of ease of use for Sunrise. Chapter 14 gives an outline of our plans for future research in this area. Finally, Chapter 15 presents our conclusions.

Part II

Results

CHAPTER 5

Sunrise

"They will speak with new tongues."

— Mark 16:17

"A wholesome tongue is a tree of life,

But perverseness in it breaks the spirit."

— Proverbs 15:4

In this chapter we describe the Sunrise programming language and its associated assertion language, which is the language studied in this work. This is a representative member of the family of imperative programming languages, and its constructs will be generally familiar to programmers. We have carefully chosen the constructs included to have natural, straightforward, and simple semantics, which will support proofs of correctness. To this end, we have extended the normal notation for some constructs, notably **while** loops and procedure declarations, to include annotations used in constructing the proof of a Sunrise program's correctness. These annotations are required, but have no effect on the actual operational semantics of the constructs involved. They could therefore be considered comments, except that they are used by the verification condition generator in order to produce an appropriate set of verification conditions to complete the proof.

In the past, there has been considerable debate over the need for the programmer to provide, say, a loop invariant. Some have claimed that this is an unreasonable burden on the programmer, who should have to provide only a program and an input/output specification. Others have replied that the requirement to provide a loop invariant forces clear thinking and documentation that should have been done in any case.

We would like to take the pragmatic position that the provision of loop invariants is necessary for the simple definition of verification condition generators, which are not complex functions. The same principle holds for the more complex annotations we require for procedures, that the provision of these annotations are necessary for simple and clean definitions of the program logic rules which serve as an axiomatic semantics for procedures. If one wishes to transfer the burden of coming up with the loop invariant from the human to the automatic computer, one incurs a great increase in the degree of difficulty of constructing the verification condition generator, including the need for automatic theorem provers, multiple decision procedures, and search strategies which have exponential time complexity. We wish to attempt something rather more tractable, and to perform only part of the task, in particular that part which seems most amenable to automatic analysis. This desire has guided the construction of the language here defined.

`exp:`	e	$::=$	$n \mid x \mid {+}{+}x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$
`(exp)list:`	es	$::=$	$\langle\,\rangle \mid CONS\ e\ es$
`bexp:`	b	$::=$	$e_1 = e_2 \mid e_1 < e_2 \mid es_1 \ll es_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b$
`cmd:`	c	$::=$	**skip** $\mid$ **abort** $\mid$ $x := e$ $\mid$ $c_1\ ;\ c_2$ $\mid$ **if** b **then** c_1 **else** c_2 **fi** $\mid$ **assert** a **with** a_{pr} **while** b **do** c **od** $\mid$ $p(x_1,\ \ldots,\ x_n\ ;\ e_1,\ \ldots,\ e_m)$
`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**

Table 5.1: Sunrise programming language.

5.1 Programming Language Syntax

Table 5.1 contains the concrete syntax of the Sunrise programming language, defined using Backus-Naur Form as a context-free grammar.

We define six types of phrases in this programming language (Table 5.2):

Type	Description	Typical Member
`exp`	numeric expressions	e
`(exp)list`	lists of numeric expressions	es
`bexp`	boolean expressions	b
`cmd`	commands	c
`decl`	declarations	d
`prog`	programs	π

Table 5.2: Sunrise programming language types of phrases.

The lexical elements of the syntax expressed in Table 5.1 are numbers and variables. Numbers (denoted by n) are simple unsigned decimal integers, including zero, with no *a priori* limit on size, to match the HOL type `num`. They cannot be negative, either as written or as the result of calculations.

Variables (denoted with x or y, etc.) are a new concrete datatype `var` consisting of two components, a string and a number. In HOL a character string may be of any length from zero or more. The name of a variable is typically printed as the string, followed immediately by the variant number, unless it is zero, when no number is printed; the possibility exists for ambiguity of the result. The parser we have constructed expects the name of the variable to consist of letters, digits, and underscore ('_'), except that the first character may also be a caret ('^'). However, the operations of the VCG allow the string to contain any characters. The meaning of the string is to be the base of the name of the variable, and the meaning of the number is to be the variant number of the variable. Hence there might be several variables with the same string but differing in their number attribute, and these are considered distinct variables. This structure is used for variables to ease the construction of variants of variables, by simply changing (increasing) the variant number of the variable.

Variables are divided into two classes, depending on the initial character (if any) of the string. If the initial character is a caret ('^'), then the variable is a *logical variable*, otherwise it is a *program variable*. Program and logical variables are completely disjoint; "`y`" and "`^y`" are separate and distinct variables. Both kinds are permitted in assertion language expressions, but only program variables are permitted in programming language expressions. Since logical variables cannot appear in programming language expressions, they may never be altered by program control, and thus retain their values unchanged throughout a computation.

The syntax given in Table 5.1 uses standard notations for readability. The actual data types (except for lists) are created in HOL as new concrete recursive datatypes, using Melham's type definition package [GM93]. The results of this definition includes the creation of the constructor functions for the various programming language syntactic phrases in Table 5.3. This forms the abstract syntax of the Sunrise programming language.

All the internal computation of the verification condition generator is based on manipulating expressions which are trees of these constructor functions and the corresponding ones for assertion language expressions. These trees are not highly legible. However, we have provided parsers and pretty-printing functions

`exp` :	$NUM\ n$	n
	$PVAR\ x$	x
	$INC\ x$	$++x$
	$PLUS\ e_1\ e_2$	$e_1 + e_2$
	$MINUS\ e_1\ e_2$	$e_1 - e_2$
	$MULT\ e_1\ e_2$	$e_1 * e_2$
`bexp` :	$EQ\ e_1\ e_2$	$e_1 = e_2$
	$LESS\ e_1\ e_2$	$e_1 < e_2$
	$LLESS\ es_1\ es_2$	$es_1 \ll es_2$
	$AND\ b_1\ b_2$	$b_1 \wedge b_2$
	$OR\ b_1\ b_2$	$b_1 \vee b_2$
	$NOT\ b$	$\sim b$
`cmd` :	$SKIP$	**skip**
	$ABORT$	**abort**
	$ASSIGN\ x\ e$	$x := e$
	$SEQ\ c_1\ c_2$	$c_1\ ;\ c_2$
	$IF\ b\ c_1\ c_2$	**if** b **then** c_1 **else** c_2 **fi**
	$WHILE\ a\ pr\ b\ c$	**assert** a **with** pr **while** b **do** c **od**
	$CALL\ p\ xs\ es$	$p(xs; es)$
`decl` :	$PROC\ p\ vars\ vals\ glbs$ $pre\ post\ calls\ rec\ c$	**proc** $p\ vars\ vals\ glbs$ $pre\ post\ calls\ rec\ c$
	$DSEQ\ d_1\ d_2$	$d_1\ ;\ d_2$
	$DEMPTY$	**empty**
`prog` :	$PROG\ d\ c$	**program** $d; c$ **end program**

Table 5.3: Sunrise programming language constructor functions.

to provide an interface that is more human-readable, so that the constructor trees are not seen for most of the time.

5.2 Informal Semantics of Programming Language

The constructs in the Sunrise programming language, shown in Table 5.1, are mostly standard. The full semantics of the Sunrise language will be given as a structural operational semantics later in this chapter. But to familiarize the reader with these constructs in a more natural and understandable way, we here give informal descriptions of the semantics of the Sunrise language. This is intended to give the reader the gist of the meaning of each operator and clause in Table 5.1. We also describe the significance of the system of annotations for both partial and total correctness.

5.2.1 Numeric Expressions

n is an unsigned integer.

x is a program variable. It may not here be a logical variable.

$++x$ denotes the increment operation, where x is a program variable as above. The increment operation adds one to the variable, stores that new value into the variable, and yields the new value as the result of the expression.

The addition, subtraction, and multiplication operators have their normal meanings, except that subtraction is restricted to nonnegative values, so $x - y = 0$ for $x \leq y$. The two operands of a binary operator are evaluated in order from left to right, and then their values are combined and the numeric result yielded.

5.2.2 Lists of Numeric Expressions

HOL provides a polymorphic list type, and a set of list operators that function on lists of any type. This list type has two constructors, NIL and $CONS$, with the standard meanings. In both its meta language and object language, HOL typically displays lists using a more compact notation, using square brackets ([ ]) to delimit lists and semicolons (;) to separate list elements. Thus $NIL = [\,]$, and [2;3;5;7] is the list of the first four primes. In this programming language we wish to reserve square brackets to denote total correctness specifications, and so we will use angle brackets ($\langle\,\rangle$) instead to denote lists within the Sunrise language, for example $\langle 2; 3; 5; 7\rangle$ or $\langle\,\rangle$. When dealing with HOL lists, however, the square brackets will still be used.

The numeric expressions in a list are evaluated in order from left to right, and their values are combined into a list of numbers which is the result yielded.

5.2.3 Boolean Expressions

The operators provided here have their standard meaning, except for $es_1 \ll es_2$, which evaluates two lists of expressions and compares their values according to their lexicographic ordering. Here the left-most elements of each list are compared first, and if the element from es_1 is less, then the test is true; if the element from es_1 is greater, then the test is false; and if the element from es_1 is the same as (equal to) the element from es_2, then these elements are discarded and the tails of es_1 and es_2 are compared in the same way, recursively.

For every operator here, the operands are evaluated in order from left to right, and their values combined and the boolean result yielded.

5.2.4 Commands

The **skip** command has no effect on the state. **abort** causes an immediate abnormal termination of the program. $x := e$ evaluates the numeric expression e and assigns the value to the variable x, which must be a program variable. $c_1 ; c_2$ executes command c_1 first, and if it terminates, then executes c_2. The conditional command **if** b **then** c_1 **else** c_2 **fi** first evaluates the boolean expression b; if it is true, then c_1 is executed, otherwise c_2 is executed.

The iteration command **assert** a **with** a_{pr} **while** b **do** c **od** evaluates b; if it is true, then the body c is executed, followed by executing the whole iteration command again, until b evaluates to false, when the loop ends. The "**assert** a" and "**with** a_{pr}" phrases of the iteration command do not affect its execution; these are here as annotations to aid the verification condition generator. a denotes an *invariant*, a condition that is true every time control passes through the head of the loop. This is used in proving the partial correctness of the loop.

In contrast, a_{pr} denotes a *progress expression*, which here must be of the form $v < x$, where v is a assertion language numeric expression and x is a logical variable. v may only contain program variables. Assertion language expressions will be defined presently; here, v serves as a *variant*, an expression whose value strictly decreases every time control passes through the head of the loop. This is used in proving the termination of the loop. In future versions of the Sunrise programming language, we intend to broaden a_{pr} to other expressions, such as $vs \ll xs$, whose variants describe values of well-founded sets.

Finally, $p(x_1, \ldots, x_n ; e_1, \ldots, e_m)$ denotes a procedure call. This first evaluates the actual value parameters $e_1, \ldots, e_m$ in order from left to right, and then calls procedure p with the resulting values and the actual variable parameters $x_1, \ldots, x_n$. The value parameters are passed by value; the variable parameters are passed by name, to simulate call-by-reference. The call must match the declaration of p in the number of both variable and value parameters. Aliasing is forbidden, that is, the actual variable parameters $x_1, \ldots, x_n$ may not contain any duplicates, and may not duplicate any global variables accessible from p. The body of p has the actual variable parameters substituted for the formal variable parameters. This substituted body is then executed on the state where the values from the actual value parameters have been bound to the formal value parameters. If the body terminates, then at the end the values of the formal value parameters are restored to their values before the procedure was entered. The effect of the procedure call is felt in the actual variable parameters and in the globals affected.

5.2.5 Declarations

The main kind of declaration is the procedure declaration; the other forms simply serve to create lists of procedure declarations or empty declarations. The procedure declaration has the concrete syntax

$$
\begin{array}{l}
\textbf{procedure}\ p\ (\textbf{var}\ x_1,\ \ldots,\ x_n\ ;\ \textbf{val}\ y_1,\ \ldots,\ y_m);\\
\quad \textbf{global}\ z_1,\ \ldots,\ z_k;\\
\quad \textbf{pre}\ a_{pre};\\
\quad \textbf{post}\ a_{post};\\
\quad \textbf{calls}\ p_1 \textbf{with}\ a_1;\\
\qquad\quad \vdots\\
\quad \textbf{calls}\ p_j \textbf{with}\ a_j;\\
\quad \textbf{recurses with}\ a_{rec};\\
\quad c\\
\textbf{end procedure}
\end{array}
$$

This syntax is somewhat large and cumbersome to repeat; we will usually use instead the lithe abstract syntax version

$$\textbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c$$

where it is to be understood that we mean

$$
\begin{array}{rcl}
p & = & p\\
vars & = & x_1,\ \ldots,\ x_n\\
vals & = & y_1,\ \ldots,\ y_m\\
vars & = & z_1,\ \ldots,\ z_k\\
pre & = & a_{pre}\\
post & = & a_{post}\\
calls & = & (\lambda p.\ \textbf{false})[a_j/p_j]\ldots[a_1/p_1]\\
rec & = & a_{rec}\\
c & = & c
\end{array}
$$

Note that the *calls* parameter is now a *progress environment* of type `prog_env`, where `prog_env` $=$ `string` $\rightarrow$ `aexp`, a function from procedure names to progress expressions, to serve as the collection of all the **calls** ... **with** phrases given.

The meaning of each one of these parameters is as follows:

- p is the name of the procedure, a simple string.
- $vars$ is the list of the formal variable parameters, a list of variables. If there are no formal variable parameters, the entire "**var** $x_1, \ldots, x_n$" phrase may be omitted.
- *vals* is the list of the formal value parameters, a list of variables. If there are no formal value parameters, the entire "**val** $y_1, \ldots, y_m$" phrase may be omitted.
- *glbs* is the list of the global variables accessible from this procedure. This includes not only those variables read or written within the body of this procedure, but also those read or written by any procedure called immediately or eventually by the body of this procedure. Thus it is a list of all globals which can possibly be read or written during the course of execution of the body once entered. If there are no globals accessible, the entire "**global** $z_1, \ldots, z_k$;" phrase may be omitted.
- *pre* is the precondition of this procedure. This is a boolean expression in the assertion language, which denotes a requirement that must be true whenever the procedure is entered. Only program variables may be used.
- *post* is the postcondition of this procedure. This is a boolean expression in the assertion language, which denotes the relationship between the states at the entrance and exit of this procedure. Two kinds of variables may be used in *post*, program variables and logical variables. The logical variables

will denote the values of variables at the time of entrance, and the program variables will denote the values of the variables at the time of exit. The postcondition expresses the logical relationship between these two sets of values, and thus describes the effect of calling the procedure.

- *calls* is the progress environment, the collection of all the **calls** ... **with** phrases given. Each "**calls** p_i **with** a_i" phrase expresses a relationship between two states, similar to the *post* expression but for different states. The first state is that at the time of entrance of this procedure p. The second state is that at any time that procedure p_i is called directly from the body of p. That is, if while executing the body of p there is a direct call to p_i, then the second state is that just after entering p_i.

  Expression a_i is a *progress expression*. Similar to the *post* expression, there are two kinds of variables that may be used in a_i, program variables and logical variables. The logical variables will denote the values of variables at the time of entrance of p, and the program variables will denote the values of the variables at the time of entrance of p_i. The progress expression gives the logical relationship between these two sets of values, and thus describes the degree of progress achieved between these calls.

- *rec* is the *recursion expression* for this procedure. It is a progress expression, similar to the progress expression of an iteration command, describing a relationship between two states. For *rec*, the first state is that at the time of entrance of p, and the second state is any time of entrance of p recursively as part of executing the body of p for the first call.

  Similar to the *post* expression, there are two kinds of variables that may be used in rec, program variables and logical variables. The logical variables will denote the values of variables at the time of original entrance of p, and the program variables will denote the values of the variables at the times of recursive entrance of p. The rec expression gives the logical relationship between these two sets of values, and thus describes the degree of progress achieved between recursive calls.

There are two permitted forms for rec. rec may be of the form $v < x$, where v is an assertion language numeric expression and x is a logical variable, or rec may be **false**. **false** is appropriate when the procedure p is not recursive and cannot call itself. Otherwise, $v < x$ should be used. v may only contain program variables; it serves as a variant, an expression whose value strictly decreases for each recursive call. Thus if v was equal to x at the time rec was originally called, then at any recursive call to p nested within that first call to p, we should have $v < x$.

In the future we intend to broaden this to include other expressions, such as $vs \ll xs$, whose variants describe values in well-founded sets, and the strict decrease described will be in terms of the relation used, e.g., $\ll$.

If this procedure is not expected to ever call itself recursively, then the phrase "**recurses with** a_{rec};" may be omitted, in which case rec is taken by default to be **false**.

- Command c is the body of this procedure. It may only use variables appearing in $vars$, $vals$, or $glbs$.

The actual significance of the various annotations, especially $calls$ and rec, will be explained in greater depth and illustrated with examples in later chapters.

5.2.6 Programs

A program consists of a declaration of a set of procedures and a command as the main body. The declarations are processed to create a *procedure environment* ρ of type `env`, collecting all of the information declared for each procedure into a function from procedure names to tuples of the following form:

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

The definition of `env` is

$$\texttt{env} \ = \ \texttt{string} \ \rightarrow \ ((\texttt{var})\texttt{list} \times (\texttt{var})\texttt{list} \times (\texttt{var})\texttt{list} \times \texttt{aexp} \times \texttt{aexp} \times \texttt{prog\_env} \times \texttt{aexp} \times \texttt{cmd}).$$

This environment is the context used for executing the bodies of the procedures themselves, and also for executing the main body of the program.

The program is considered to begin execution in a state where the value of all variables is zero; however, this initial state is not included in the proof of a program's correctness. A future version of the Sunrise program may have an arbitrary initial state, and the same programs will prove correct.

5.3 Assertion Language Syntax

Table 5.4 contains the syntax of the Sunrise assertion language, defined using Backus-Naur Form as a context-free grammar.

We define three types of phrases in this assertion language, in Table 5.5.

The above syntax uses standard notations for readability. The actual data types are created in HOL as new concrete recursive datatypes, using Melham's

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

Table 5.4: Sunrise assertion language.

Type	Description	Typical Member
`vexp`	numeric expressions	v
`(vexp)list`	lists of numeric expressions	vs
`aexp`	boolean expressions	a

Table 5.5: Sunrise assertion language types of phrases.

type definition package [GM93]. The results of this definition includes the creation of the constructor functions for the various assertion language syntactic phrases in Table 5.6. This forms the abstract syntax of the Sunrise assertion language.

`vexp` :	$ANUM\ n$	n
	$AVAR\ x$	x
	$APLUS\ v_1\ v_2$	$v_1 + v_2$
	$AMINUS\ v_1\ v_2$	$v_1 - v_2$
	$AMULT\ v_1\ v_2$	$v_1 * v_2$
`aexp` :	$ATRUE$	**true**
	$AFALSE$	**false**
	$AEQ\ v_1\ v_2$	$v_1 = v_2$
	$ALESS\ v_1\ v_2$	$v_1 < v_2$
	$ALLESS\ vs_1\ vs_2$	$vs_1 \ll vs_2$
	$AAND\ a_1\ a_2$	$a_1 \wedge a_2$
	$AOR\ a_1\ a_2$	$a_1 \vee a_2$
	$ANOT\ a$	$\sim a$
	$AIMP\ a_1\ a_2$	$a_1 \Rightarrow a_2$
	$AEQB\ a_1\ a_2$	$a_1 = a_2$
	$ACOND\ a_1\ a_2\ a_3$	$a_1 => a_2 \mid a_3$
	$ACLOSE\ a$	**close** a
	$AFORALL\ x\ a$	$\forall x.\ a$
	$AEXISTS\ x\ a$	$\exists x.\ a$

Table 5.6: Sunrise assertion language constructor functions.

5.4 Informal Semantics of Assertion Language

The constructs in the Sunrise assertion language, shown in Table 5.4, are mostly standard. The full semantics of the Sunrise assertion language will be given as a denotational semantics later in this chapter. But to familiarize the reader with these constructs in a more natural and understandable way, we here give informal descriptions of the semantics of the Sunrise assertion language. This is intended to give the reader the gist of the meaning of each operator and clause.

The evaluation of any expression in the assertion language cannot change the state; hence it is immaterial in what order subexpressions are evaluated.

5.4.1 Numeric Expressions

n is an unsigned integer, as before for the programming language.

x is a variable, which may be either a program variable or a logical variable.

The addition, subtraction, and multiplication operators have their normal meanings, except that subtraction is restricted to nonnegative values, so $x - y = 0$ for $x \leq y$.

5.4.2 Lists of Numeric Expressions

These are similar to the lists of numeric expressions described previously for the programming language, except that the constituent expressions are assertion language numeric expressions. This list type has two constructors, NIL and $CONS$, with the standard meanings.

5.4.3 Boolean Expressions

Most of the operators provided here have their standard meaning, and are similar to their counterparts in the programming language, if one exists. **true** and

false are the logical constants. $=$ and $<$ have the normal interpretation, and so do the various boolean operators, such as conjunction ($\wedge$) and disjunction ($\vee$). $vs_1 \ll vs_2$ evaluates two lists of expressions and compares their values according to their lexicographic ordering. $(a_1 => a_2 \mid a_3)$ is a conditional expression, first evaluating a_1, and then yielding the value of a_2 or a_3 respectively, depending on whether a_1 evaluated to T or F, which are the HOL truth constants. **close** a forms the universal closure of a, which is true when a is true for all possible assignments to its free variables. We have specifically included the universal and existential quantifiers; all quantification is over the nonnegative integers.

5.5 Formal Semantics

> "There are, it may be, so many kinds of languages in the world, and none of them is without significance. Therefore, if I do not know the meaning of the language, I shall be a foreigner to him who speaks, and he who speaks will be a foreigner to me."
> — 1 Corinthians 14:10, 11

We present in this section the structural operational semantics of the Sunrise programming language, according to the style of Plotkin [Plo81] and Hennessey [Hen90]. We also present the semantics of the Sunrise assertion language in a denotational style.

The definitions in this section are the primary foundation for all succeeding proof activity. In particular, it is from these definitions that the five program logics described in Chapter 6 are proven sound, and from which the verification condition generator presented in Chapter 7 is proven sound. It is therefore also the foundation for the example programs which are verified in Chapter 8.

These extensions to the HOL system are purely definitional. No new axioms are asserted. This is therefore classified as a "conservative extension" of HOL, and there is no possibility of unsoundness entering the system. This security was essential to our work. This choice ensured that we faced a very difficult task in proving the soundness of the logics of Chapter 6, and in fact this may have consumed 65–70% of the effort of this project. These proofs culminated in the VCG soundness theorems, and once proven, the theorems are applied to example programs without needing to retrace the same proofs for each example.

This significant expenditure of effort was necessary because of the history of unsoundness in proposed axiomatic semantics, particularly in relation to procedures. After constructing the necessary proofs, we are grateful for the unrelenting vigilance of the HOL system, which kept us from proving any incorrect theorems. Apparently it is easier to formulate a correct structural operational semantics than it is to formulate a sound axiomatic semantics. This agrees with our intuition, that an axiomatic semantics is inherently higher-level than operational semantics, and omits details covered at the lower level. We exhibit this structural operational semantics as the critical foundation for our work, and present it for the research community's appraisal.

As previously described, the programming language has six kinds of phrases, and the assertion language has three. For each programming language phrase, we define a relation to denote the semantics of that phrase. The structural operational semantics consists of a series of rules which together constitute an inductive definition of the relation. This is implemented in HOL using Melham's excellent library [Mel91] for inductive rule definitions.

The semantics of the assertion language is defined in a denotational style. For each assertion language phrase, we define a function which yields the interpretation of that phrase into the HOL Object Language. This is implemented in HOL using Melham's tool for defining recursive functions on concrete recursive types [Mel89]. The types used here are the types of the assertion language phrases.

5.5.1 Programming Language Structural Operational Semantics

The structural operational semantics of the six kinds of Sunrise programming language phrases is expressed by the six relations in Table 5.7.

$E\ e\ s_1\ n\ s_2$	numeric expression e:`exp` evaluated in state s_1 yields numeric value n:`num` and state s_2
$ES\ es\ s_1\ ns\ s_2$	numeric expressions es:`(exp)list` evaluated in state s_1 yield numeric values ns:`(num)list` and state s_2
$B\ b\ s_1\ t\ s_2$	boolean expression b:`bexp` evaluated in state s_1 yields truth value t:`bool` and state s_2
$C\ c\ \rho\ s_1\ s_2$	command c:`cmd` evaluated in environment ρ and state s_1 yields state s_2
$D\ d\ \rho_1\ \rho_2$	declaration d:`decl` elaborated in environment ρ_1 yields result environment ρ_2
$P\ \pi\ s$	program π:`prog` executed yields state s

Table 5.7: Sunrise programming language semantic relations.

In Table 5.8, we present rules that inductively define the numeric expression semantic relation E. This is a structural operational semantics for numeric expressions.

Number:

$$\frac{}{E\ (n)\ s\ n\ s}$$

Variable:

$$\frac{}{E\ (x)\ s\ (s\ x)\ s}$$

Increment:

$$\frac{E\ (x)\ s_1\ n\ s_2}{E\ (++x)\ s_1\ (n+1)\ s_2[(n+1)/x]}$$

Addition:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1+e_2)\ s_1\ (n_1+n_2)\ s_3}$$

Subtraction:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1-e_2)\ s_1\ (n_1-n_2)\ s_3}$$

Multiplication:

$$\frac{\begin{array}{c}E\ e_1\ s_1\ n_1\ s_2\\ E\ e_2\ s_2\ n_2\ s_3\end{array}}{E\ (e_1*e_2)\ s_1\ (n_1*n_2)\ s_3}$$

Table 5.8: Numeric Expression Structural Operational Semantics.

In Table 5.9, we present rules that inductively define the numeric expression list semantic relation ES. This is a structural operational semantics for lists of numeric expressions. The ES relation was actually defined in HOL as a list

Nil:

$$\frac{}{ES\ (\langle\rangle)\ s\ []\ s}$$

Cons:

$$\frac{E\ e\ s_1\ n\ s_2 \qquad ES\ es\ s_2\ ns\ s_3}{ES\ (CONS\ e\ es)\ s_1\ (CONS\ n\ ns)\ s_3}$$

Table 5.9: Numeric Expression List Structural Operational Semantics.

recursive function, with two cases for the definition based on NIL or $CONS$.

In Table 5.10, we present rules that inductively define the boolean expression semantic relation B. This is a structural operational semantics for boolean expressions.

Equality:

$$\frac{E\ e_1\ s_1\ n_1\ s_2 \qquad E\ e_2\ s_2\ n_2\ s_3}{B\ (e_1 = e_2)\ s_1\ (n_1 = n_2)\ s_3}$$

Less Than:

$$\frac{E\ e_1\ s_1\ n_1\ s_2 \qquad E\ e_2\ s_2\ n_2\ s_3}{B\ (e_1 < e_2)\ s_1\ (n_1 < n_2)\ s_3}$$

Lexicographic Less Than:

$$\frac{ES\ es_1\ s_1\ ns_1\ s_2 \qquad ES\ es_2\ s_2\ ns_2\ s_3}{B\ (es_1 \ll es_2)\ s_1\ (ns_1 \ll ns_2)\ s_3}$$

Conjunction:

$$\frac{B\ b_1\ s_1\ t_1\ s_2 \qquad B\ b_2\ s_2\ t_2\ s_3}{B\ (b_1 \wedge b_2)\ s_1\ (t_1 \wedge t_2)\ s_3}$$

Disjunction:

$$\frac{B\ b_1\ s_1\ t_1\ s_2 \qquad B\ b_2\ s_2\ t_2\ s_3}{B\ (b_1 \vee b_2)\ s_1\ (t_1 \vee t_2)\ s_3}$$

Negation:

$$\frac{B\ b\ s_1\ t\ s_2}{B\ (\sim b)\ s_1\ (\sim t)\ s_2}$$

Table 5.10: Boolean Expression Structural Operational Semantics.

In Table 5.11, we present rules that inductively define the command semantic relation C. This is a structural operational semantics for commands.

Skip:

$$\frac{}{C\ \mathbf{skip}\ \rho\ s\ s}$$

Abort:

(no rules)

Assignment:

$$\frac{E\ e\ s_1\ n\ s_2}{C\ (x := e)\ \rho\ s_1\ s_2[n/x]}$$

Sequence:

$$\frac{C\ c_1\ \rho\ s_1\ s_2, \qquad C\ c_2\ \rho\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ \rho\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c_1\ \rho\ s_2\ s_3}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ \rho\ s_1\ s_3}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2, \qquad C\ c_2\ \rho\ s_2\ s_3}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ \rho\ s_1\ s_3}$$

Iteration:

$$\frac{\begin{array}{l} B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c\ \rho\ s_2\ s_3 \\ C\ (\mathbf{assert}\ a\ \mathbf{with}\ a_{pr} \\ \qquad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ s_3\ s_4 \end{array}}{\begin{array}{l} C\ (\mathbf{assert}\ a\ \mathbf{with}\ a_{pr} \\ \qquad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ s_1\ s_4 \end{array}}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2}{\begin{array}{l} C\ (\mathbf{assert}\ a\ \mathbf{with}\ a_{pr} \\ \qquad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ s_1\ s_2 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c \rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \\ C\ (c \lhd [xs\ \&\ vals'/vars\ \&\ vals])\ \rho\ s_2[ns/vals']\ s_3 \end{array}}{C\ (\mathbf{call}\ p(xs;\ es))\ \rho\ s_1\ s_3[(\mathbf{map}\ s_2\ vals')/vals']}$$

Table 5.11: Command Structural Operational Semantics.

In Table 5.12, we present rules that inductively define the declaration semantic relation D. This is a structural operational semantics for declarations.

Procedure Declaration:

$$\frac{}{\begin{array}{r} D\ (\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c)\ \rho \\ \rho[\langle vars, vals, glbs, pre, post, calls, rec, c\rangle / p] \end{array}}$$

Declaration Sequence:

$$\frac{D\ d_1\ \rho_1\ \rho_2, \qquad D\ d_2\ \rho_2\ \rho_3}{D\ (d_1\ ;\ d_2)\ \rho_1\ \rho_3}$$

Empty Declaration:

$$\frac{}{D\ (\mathbf{empty})\ \rho\ \rho}$$

Table 5.12: Declaration Structural Operational Semantics.

In Table 5.13, we present rules that inductively define the program semantic relation P. This is a structural operational semantics for programs. As used in this definition, we define ρ_0 as the empty environment

$$\rho_0 = \lambda p.\ \langle [\,],\ [\,],\ [\,],\ \mathbf{false},\ \mathbf{true},\ (\lambda p.\ \mathbf{false}),\ \mathbf{false},\ \mathbf{abort}\rangle,$$

and s_0 as the initial state $s_0 = \lambda x.\ 0$.

Program:

$$\frac{D\ d\ \rho_0\ \rho_1, \qquad C\ c\ \rho_1\ s_0\ s_1}{P\ (\mathbf{program}\ d;\ c\ \mathbf{end\ program})\ s_1}$$

Table 5.13: Program Structural Operational Semantics.

5.5.2 Assertion Language Denotational Semantics

The denotational semantics of the three kinds of Sunrise assertion language phrases is expressed by the three functions in Table 5.14.

$V\ v\ s$	numeric expression v:`vexp` evaluated in state s
	yields numeric value in `num`
$VS\ vs\ s$	list of numeric expressions vs:`(vexp)list` evaluated in state s
	yields list of numeric values in `(num)list`
$A\ a\ s$	boolean expression a:`aexp` evaluated in state s
	yields truth value in `bool`

Table 5.14: Sunrise assertion language semantic functions.

In Table 5.15, we present a denotational definition of the assertion language semantic function V for numeric expressions.

$$
\begin{array}{lcl}
V\ n\ s & = & n \\
V\ x\ s & = & s\ x \\
V\ (v_1 + v_2)\ s & = & V\ v_1\ s + V\ v_2\ s \\
V\ (v_1 - v_2)\ s & = & V\ v_1\ s - V\ v_2\ s \\
V\ (v_1 * v_2)\ s & = & V\ v_1\ s * V\ v_2\ s
\end{array}
$$

Table 5.15: Assertion Numeric Expression Denotational Semantics.

In Table 5.16, we present a denotational definition of the assertion language semantic function VS for lists of numeric expressions.

$$
\begin{array}{lcl}
VS\ \langle\rangle\ s & = & [\,] \\
VS\ (CONS\ v\ vs)\ s & = & CONS\ (V\ v\ s)\ (VS\ vs\ s)
\end{array}
$$

Table 5.16: Assertion Numeric Expression List Denotational Semantics.

In Table 5.17, we present a denotational definition of the assertion language semantic function A for boolean expressions.

$$
\begin{array}{lcl}
A\ \mathbf{true}\ s & = & \mathrm{T} \\
A\ \mathbf{false}\ s & = & \mathrm{F} \\
A\ (v_1 = v_2)\ s & = & (V\ v_1\ s = V\ v_2\ s) \\
A\ (v_1 < v_2)\ s & = & (V\ v_1\ s < V\ v_2\ s) \\
A\ (vs_1 \ll vs_2)\ s & = & (VS\ vs_1\ s \ll VS\ vs_2\ s) \\
A\ (a_1 \wedge a_2)\ s & = & (A\ a_1\ s \wedge A\ a_2\ s) \\
A\ (a_1 \vee a_2)\ s & = & (A\ a_1\ s \vee A\ a_2\ s) \\
A\ (\sim a)\ s & = & \sim(A\ a\ s) \\
A\ (a_1 \Rightarrow a_2)\ s & = & (A\ a_1\ s \Rightarrow A\ a_2\ s) \\
A\ (a_1 = a_2)\ s & = & (A\ a_1\ s = A\ a_2\ s) \\
A\ (a_1 => a_2 \mid a_3)\ s & = & (A\ a_1\ s => A\ a_2\ s \mid A\ a_3\ s) \\
A\ (\mathbf{close}\ a)\ s & = & (\forall s_1.\ A\ a\ s_1) \\
A\ (\forall x.\ a)\ s & = & (\forall n.\ A\ a\ s[n/x]) \\
A\ (\exists x.\ a)\ s & = & (\exists n.\ A\ a\ s[n/x])
\end{array}
$$

Table 5.17: Assertion Boolean Expression Denotational Semantics.

The lexicographic ordering $\ll$ is defined as

$$
\begin{array}{rcl}
[\,] \ll [\,] & = & \mathrm{F} \\
[\,] \ll CONS\ n\ ns & = & \mathrm{T} \\
CONS\ n\ ns \ll [\,] & = & \mathrm{F} \\
CONS\ n_1\ ns_1 \ll CONS\ n_2\ ns_2 & = & n_1 < n_2\ \vee\ (n_1 = n_2\ \wedge\ ns_1 \ll ns_2)
\end{array}
$$

This concludes the definition of the semantics of the assertion language.

The Sunrise language is properly thought of as consisting of both the programming language *and* the assertion language, even though the assertion language is never executed, and only exists to express specifications and annotations, to facilitate proofs of correctness. The two languages are different in character; the semantics of the programming language is very dependent on time; it both responds to and causes the constantly changing state of the memory. In contrast, the assertion language has a timeless quality, where, for a given state, an expression will always evaluate to the same value irrespective of how many times it is evaluated. The variables involved also reflect this, where program variables often change their values during execution, but logical variables never do. The programming language is an active, involved participant in the execution as it progresses; the assertion language takes the role of a passive, detached observer of the process.

This difference carries over to how the languages are used. States and their changes in time are the central focus of the operational semantics, whereas assertions and their permanent logical interrelationships are the focus of the axiomatic semantics. Programs in the programming language are executed, causing changes to the state. Assertions in the assertion language are never executed or even evaluated. Instead they are stepping stones supporting the proofs of correctness, which also have a timeless quality. Done once for all possible executions of the program, a proof replaces and exceeds any finite number of tests.

5.6 Procedure Entrance Semantic Relations

In addition to the traditional structural operational semantics of the Sunrise programming language, we also define two semantic relations that connect to states reached at the entrances of procedures called from within a command. These semantic relations are used to define the correctness specifications for the Entrance Logic.

The entrance structural operational semantics of commands and procedures is expressed by the two relations described in Table 5.18.

$C\_calls\ c\ \rho\ s_1\ p\ s_2$	Command c:`cmd`, evaluated in environment ρ and state s_1, calls procedure p directly from c, where the state just after entering p is s_2.
$M\_calls\ p_1\ s_1\ ps\ p_2\ s_2\ \rho$	The body of procedure p_1, evaluated in environment ρ and state s_1, goes through a path ps of successively nested calls, and finally calls p_2, where the state just after entering p_2 is s_2.

Table 5.18: Sunrise programming language entrance semantic relations.

In Table 5.19, we present rules that inductively define the command semantic relation C_calls.

Skip:

(no rules)

Abort:

(no rules)

Assignment:

(no rules)

Sequence:

$$\frac{C\_calls\ c_1\ \rho\ s_1\ p\ s_2}{C\_calls\ (c_1\ ;\ c_2)\ \rho\ s_1\ p\ s_2}$$

$$\frac{\begin{array}{c} C\ c_1\ \rho\ s_1\ s_2 \\ C\_calls\ c_2\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (c_1\ ;\ c_2)\ \rho\ s_1\ p\ s_3}$$

Conditional:

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C\_calls\ c_1\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ p\ s_3}$$

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{F}\ s_2 \\ C\_calls\ c_2\ \rho\ s_2\ p\ s_3 \end{array}}{C\_calls\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ \rho\ s_1\ p\ s_3}$$

Iteration:

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C\_calls\ c\ \rho\ s_2\ p\ s_3 \end{array}}{\begin{array}{l} C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ p\ s_3 \end{array}}$$

$$\frac{\begin{array}{l} \quad B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C\ c\ \rho\ s_2\ s_3 \\ C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_3\ p\ s_4 \end{array}}{\begin{array}{l} C\_calls\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr} \\ \qquad\qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ \rho\ s_1\ p\ s_4 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \end{array}}{C\_calls\ (\textbf{call}\ p\ (xs; es))\ \rho\ s_1\ p((s_2[ns/vals']) \lhd [xs\ \&\ vals'/vars\ \&\ vals])}$$

Table 5.19: Command Entrance Semantic Relation.

In Table 5.20, we present rules that inductively define the procedure path semantic relation M_calls.

Single:

$$\frac{\rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \qquad C\_calls\ c\ \rho\ s_1\ p_2\ s_2}{M\_calls\ p_1\ s_1\ [\,]\ p_2\ s_2\ \rho}$$

Multiple:

$$\frac{M\_calls\ p_1\ s_1\ ps_1\ p_2\ s_2\ \rho \qquad M\_calls\ p_2\ s_2\ ps_2\ p_3\ s_3\ \rho}{M\_calls\ p_1\ s_1\ (ps_1\ \&\ (CONS\ p_2\ ps_2))\ p_3\ s_3\ \rho}$$

Table 5.20: Path Entrance Semantic Relation.

5.7 Termination Semantic Relations

In addition to the other structural operational semantics of the Sunrise programming language, we also define two semantic relations that describe the termination of executions begun in states reached at the entrances of procedures called from within a command. These semantic relations are used to define the correctness specifications for the Termination Logic.

The termination semantics of commands and procedures is expressed by the two relations in Table 5.21. These termination semantic relations are true when all direct calls from c or from the body of p_1 are known to terminate.

$C\_calls\_terminate\ c\ \rho\ s_1$	For every procedure p and state s_2 such that
	$C\_calls\ c\ \rho\ s_1\ p\ s_2$,
	the body of p executed in state s_2 terminates.
$M\_calls\_terminate\ p_1\ s_1\ \rho$	For every procedure p_2 and state s_2 such that
	$M\_calls\ p_1\ s_1\ [\,]\ p_2\ s_2\ \rho$,
	the body of p_2 executed in state s_2 terminates.

Table 5.21: Sunrise programming language termination semantic relations.

In Tables 5.22 and 5.23, we present the definitions of the command termination semantic relation *C_calls_terminate* and the procedure path termination semantic relation *M_calls_terminate*.

$$\begin{array}{l}
C_calls_terminate\ c\ \rho\ s_1\ \ = \\
\qquad \forall p\ s_2.\ C_calls\ c\ \rho\ s_1\ p\ s_2\ \Rightarrow \\
\qquad\qquad (\exists s_3.\ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c' \rangle = \rho\ p\ \mathbf{in} \\
\qquad\qquad\quad C\ c'\ \rho\ s_2\ s_3)
\end{array}$$

Table 5.22: Command Termination Semantic Relation *C_calls_terminate*.

$$\begin{array}{l}
M_calls_terminate\ p_1\ s_1\ \rho\ \ = \\
\qquad \forall p_2\ s_2.\ M_calls\ p_1\ s_1\ [\,]\ p_2\ s_2\ \rho\ \Rightarrow \\
\qquad\qquad (\exists s_3.\ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c \rangle = \rho\ p_2\ \mathbf{in} \\
\qquad\qquad\quad C\ c\ \rho\ s_2\ s_3)
\end{array}$$

Table 5.23: Procedure Path Termination Semantic Relation *M_calls_terminate*.

The definitions of the relations presented in this chapter define the semantics of the Sunrise programming language, as a foundation for all later work. From this point on, all descriptions of the meanings of program phrases will be proven as theorems from this foundation, with the proofs mechanically checked. This will ensure the soundness of the later axiomatic semantics, a necessary precondition to a verified VCG.

CHAPTER 6

Program Logics

"And you shall teach them the statutes and the laws, and show them the way in which they must walk and the work they must do."

— Exodus 18:20

"Prove all things; hold fast that which is good."

— 1 Thessalonians 5:21, King James Version

Floyd's and Hoare's seminal papers ([Flo67], [Hoa69]) set forth the idea that one could reason about all executions of a program using the axioms and rules of inference of a logic. The axioms and rules of this logic describe valid patterns of deduction, and involve both phrases of the programming language, and assertions describing conditions at points in the execution. A key element of this reasoning process is that it involves only syntactic manipulations of the program and assertion language phrases involved. This is inherently much simpler than following the same structure of reasoning by tracing the sequence of states that the computation passes through according to the operational semantics. We distinguish these two kinds of reasoning as "syntactic" versus "semantic" reasoning. Essentially, syntactic reasoning involves much simpler operations, which is a great advantage, *if* the syntactic reasoning is semantically valid. Then the syntactic reasoning step embodies and stands for a level of semantic reasoning, which only need be verified once. This then saves one from repeating the same patterns of semantic reasoning every time the syntactic manipulation applies.

In this chapter we will describe five program logics, which together constitute an axiomatic semantics for total correctness for the Sunrise programming language. These logics and their rules are the "laws" referred to in the introductory quote. Unlike previously proposed axiomatic semantics, every rule in every logic presented in this chapter is not simply asserted or proposed, but in fact has been mechanically proven correct as a theorem from the underlying structural operational semantics. Much of the content of these logics concerns proving the total correctness of mutually recursive procedures.

In the past, axiomatic semantics for total correctness for procedures has involved a rule for procedure call similar to the following rule by Sokołowski [Sok77]:

$$\frac{\begin{array}{c}\{q(0)\}\ B\ \{r\} \\ \{q(i)\}\ \mathbf{call}\ p\ \{r\}\ \vdash\ \{q(i+1)\}\ B\ \{r\}\end{array}}{\{\exists i \geq 0.\ q(i)\}\ \mathbf{call}\ p\ \{r\}}$$

The argument to q is a recursion depth counter, which must decrease by exactly one for each procedure call. Sokołowski described the need to find an appropriate meaning for the phrase

$$\{q(i)\}\ \mathbf{call}\ p\ \{r\}\ \vdash\ \{q(i+1)\}\ B\ \{r\}.$$

He then gave an interpretation which involved an infinite chain of predicate transformers.

In the various papers which have proposed rules similar to this one, the example proofs presented appeared to us to have an *ad hoc* quality, where the proof depended greatly on the specific example, and not as much on the verification mechanism. Thus the proofs of the examples seemed somewhat irregular in shape, although entirely valid.

In our investigation, we have created a new approach to the proof of total correctness of procedures not deriving from the above style of rule for procedure call. The approach we give has considerably more mechanism than the single rule above; but we find that the additional mechanism give a structure to the proof which largely removes the *ad hoc* quality, and in fact regularizes the process enough that it can be successfully mechanized in a verification condition generator. In addition, that verification condition generator then removes from the user's view all of the new mechanism, leaving only a set of relatively simple verification conditions which do not themselves involve any recursion.

This additional mechanism is an aid, in that it moves much of the proof effort out of the arena which is particular for each individual program to be proved, into the area which is regularized and structured, with established patterns of reasoning. It also helps the user in that it breaks a large problem into smaller pieces, and allows a more incremental, stepwise, "line upon line" construction of a proof.

In addition, our system appears to be more general than the previous proposals. These generally asked the user to supply a recursion depth counter that decreased by exactly one for each call. Instead of this, we ask the user to supply a recursion expression which must decrease by *at least* one every time a nested recursive call is made to the same procedure. This might be an immediately recursive call, as in the factorial procedure; or it might be an eventual recursive call, as in a top-down recursive descent parser that may have many intermediate calls between a call of a particular procedure and a recursive entry of the same procedure. This is a looser condition than previously proposed, and thus will support proofs of total correctness for a larger class of programs. We do not claim that our system can support proofs of total correctness for *all* programs which in fact terminate; there may be some exotic examples which cannot be verified within this structure that we propose. Nevertheless, seems to us at this point that our system may be expressive enough to cover most of the programs that would be written in actual practice.

This claim of generality must be qualified, however. In general, it may be possible to find a recursion depth counter that decreases by exactly one for each call for any example program which could be proven by our system. However, we agree with Pandya and Joseph [PJ86] that this can be difficult in practice because it leads to the use of predicates which are often complex and non-intuitive, even for simple programs. Pandya and Joseph make the excellent point that it is important for a program proof to make a proper use of abstraction, to remove unnecessary details from the burden imposed on the user, and to be structured in a natural, intuitive way. We wholly agree, and have constructed the system contained in this dissertation to reflect this concern for proper abstraction, natural and intuitive steps, and structuring the proof to reflect the structure present in the program itself. Our claim of generality should then be understood in the sense of this more intuitive and natural approach.

The core of our system's approach to proving the termination of recursive procedure calls uses an expression, supplied by the user in the **recurses with** part of the specification of a procedure, which we will call the *recursion expression* of that procedure. This part of the specification is a claim that the recursion expression's value decreases by at least one between recursive calls of that procedure. If this is true, then for any value that the expression may have the first time that procedure is called, it can only decrease a finite number of times, and thus must eventually come to a place where it does not call itself recursively any more. This guarantees that the procedure terminates.

To verify that the recursion expression's value decreases by at least one between recursive calls of the procedure requires that we compare the value of this expression at two different times, which may be widely separated with a chain of many nested calls in between. We break this chain down into the individual steps achieved between each procedure call in this chain and the next. The progress achieved in each individual procedure call is described in the **calls** ... **with** part of the specification of the procedure. Then the progress achieved between recursive procedure calls is the accumulation of the progress achieved in each step.

This then requires that we verify the progress claimed in the **calls** ... **with** part of the specification of the procedure. This progress specification describes the change in state between two points in time, one at the head of the procedure's body, which we call the *entrance* of the procedure, and the other at the entrance of the procedure named in the **calls** ... **with** specification. We can define this progress by a new form of program logic, described in detail below.

This new form of program logic may seem strange at first glance. The traditional Hoare logic partial correctness specification has the form $\{a_1\}\ c\ \{a_2\}\ /\rho$,

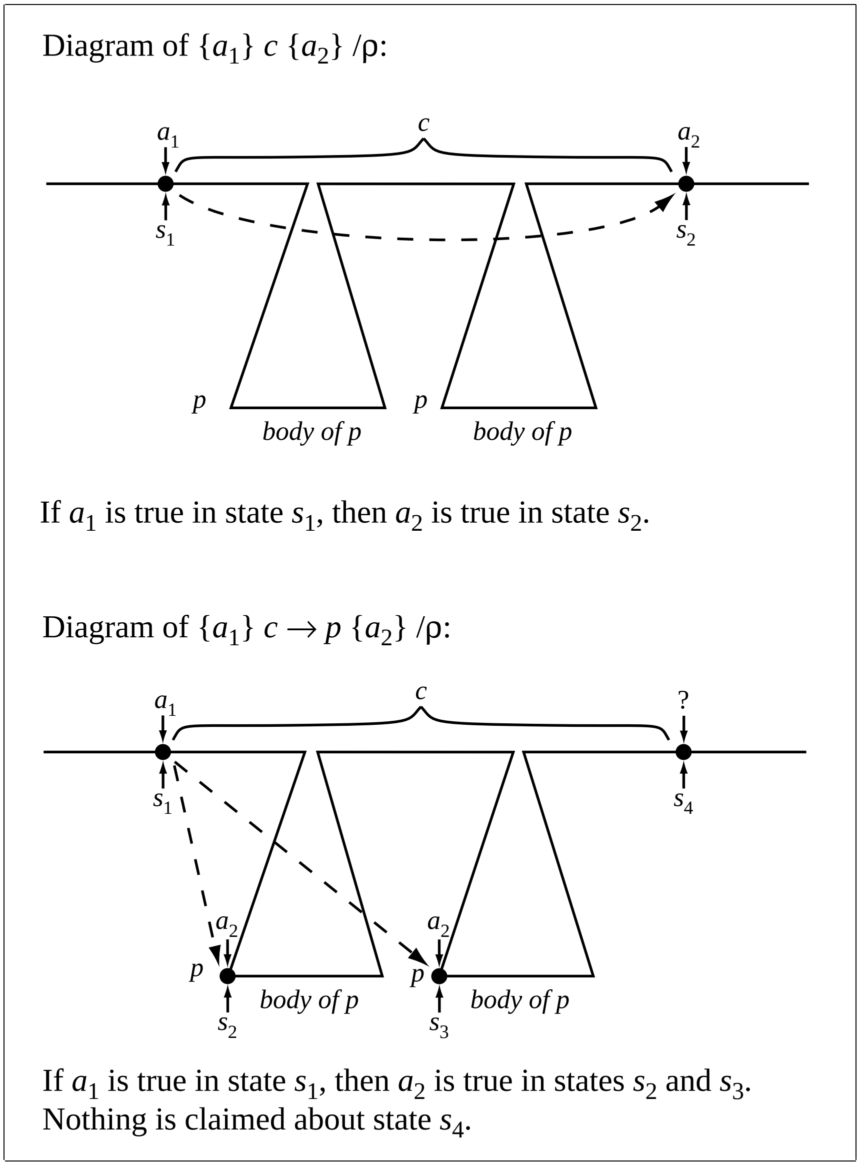

Figure 6.1: Comparison of Partial Correctness and Entrance Specifications.

describing the relationship between the states before and after executing the command c, given the procedure environment ρ. One of the new correctness specifications we propose has the form $\{a_1\}\ c \rightarrow p\ \{a_2\}\ /\rho$, describing the relationship between the states (1) before executing c and (2) just after entering the procedure p as a result of a call which issued from within the command c. Whereas the traditional correctness specification relates two points in the computation which are at the same level of procedure call, the new correctness specification relates two points which are at two different levels of procedure call. Further, where the traditional correctness specification gives a postcondition describing the state at the end of executing the command c, the new correctness specification does not in any way describe the state at the end of c, but rather the states at particular points *within* the execution of c. This is diagrammed in Figure 6.1. The traditional correctness specification is diagrammed as a horizontal dashed arrow to the right, denoting the progress of computation between the beginning of c and its end. The new correctness specification is diagrammed as a *diagonal* arrow, pointing down and to the right, denoting the progress of computation between the beginning of c and the points of entry of a procedure called directly from within c.

The purpose of this new correctness specification is to be able to express the progress achieved from the beginning of the entire body of a procedure to the points of entry of procedures called from within the body. This is used to verify the **calls** ... **with** specifications, which are then used in turn to verify the **recurses with** specifications. These are then used to prove the termination of procedures, an essential element in proving the total correctness of programs.

```
program
    procedure odd(var a;  val n);
        pre     true;
        post    (∃b. n̂ = 2 * b + a)  ∧  a < 2  ∧  n = n̂;
        calls   odd     with n < n̂;
        calls   even    with n < n̂;
        recurses        with n < n̂;

        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. n̂ + 1 = 2 * b + a)  ∧  a < 2  ∧  n = n̂;
        calls   even    with n < n̂;
        calls   odd     with n < n̂;
        recurses        with n < n̂;

        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 ]
```

Table 6.1: Odd/Even Example Program.

To make these ideas more concrete, let us take as a specific example the program in Table 6.1. This is the odd/even program. It has two mutually recursive procedures, *odd* and *even*, each of which calls itself and the other. The procedures actually could have been written with far less recursion; this version was created to exhibit as much recursion as possible. The procedure call progress expressions all declare that the value of the variable n decreases for each call, and this is the progress declared by the recursion expressions as well. This odd/even program will serve as a running example throughout this chapter to illustrate several of the correctness specifications that we describe.

6.1 Total Correctness of Expressions

In Table 6.2, we present a Hoare logic for the total correctness of numeric and boolean expressions in the Sunrise programming language. This is the first of three newly invented logics of this dissertation. It is based on three new correctness specifications, for numeric expressions, lists of numeric expressions, and boolean expressions. Generally speaking, this is a modest expression logic. We have added side effects in only one operator, the increment operator, and none of the operators are either nondeterministic or nonterminating. In the future, we intend to explore these other possibilities. This logic is intended to show a robust structure capable of growth.

The key rules are the ones for expression preconditions. The functions *ae_pre*, *aes_pre*, and *ab_pre* calculate appropriate preconditions which guarantee that the given postcondtion is true after executing the expression. The precondition is not simply the same as the postcondition, because the programming language

Precondition Strengthening:

$$\frac{\begin{array}{c}\{p \Rightarrow a\}\\ [a]\ e\ [q]\end{array}}{[p]\ e\ [q]}$$

$$\frac{\begin{array}{c}\{p \Rightarrow a\}\\ [a]\ es\ [q]\end{array}}{[p]\ es\ [q]}$$

$$\frac{\begin{array}{c}\{p \Rightarrow a\}\\ [a]\ b\ [q]\end{array}}{[p]\ b\ [q]}$$

Postcondition Weakening:

$$\frac{\begin{array}{c}[p]\ e\ [a]\\ \{a \Rightarrow q\}\end{array}}{[p]\ e\ [q]}$$

$$\frac{\begin{array}{c}[p]\ es\ [a]\\ \{a \Rightarrow q\}\end{array}}{[p]\ es\ [q]}$$

$$\frac{\begin{array}{c}[p]\ b\ [a]\\ \{a \Rightarrow q\}\end{array}}{[p]\ b\ [q]}$$

False Precondition:

$$\overline{[\mathbf{false}]\ e\ [q]}$$

$$\overline{[\mathbf{false}]\ es\ [q]}$$

$$\overline{[\mathbf{false}]\ b\ [q]}$$

Numeric Expression Precondition:

$$\overline{[ae\_pre\ \ e\ \ q]\ e\ [q]}$$

Expression List Precondition:

$$\overline{[aes\_pre\ \ es\ \ q]\ es\ [q]}$$

Boolean Expression Precondition:

$$\overline{[ab\_pre\ \ b\ \ q]\ b\ [q]}$$

Table 6.2: General Rules for Total Correctness of Expressions.

we are considering allows expressions to have side effects, and this change of state requires a change in the expression that describes the state. For a complete definition of the functions *ae_pre*, *aes_pre*, and *ab_pre*, see the Section 10.3 on Translations.

In Tables 6.3, 6.4, and 6.5, we have the rules of inference for individual expressions in the Sunrise programming language. All of these in fact are subsumed by the three rules in Table 6.2 for expression preconditions, but are presented here for completeness.

6.1.1 Closure Specification

$$\{a\}$$

$$a \quad : \quad \text{assertion language condition}$$

6.1.1.1 Semantics of Closure Specification

$$\{a\} \;=\; (\forall s.\ A\ a\ s)$$

Assertion language boolean expression a is true in every state, and thus is equivalent to the universal closure of a. These expressions are deterministic, have no side effects, and always terminate.

These should not be confused with partial correctness specifications, for example $\{p\}\ c\ \{q\}\ /\rho$. The $\{p\}$ and $\{q\}$ in the partial correctness specifications do not refer to closure specifications, but to conditions about two different states at the beginning and end of executions of the command c. In contrast, closure specifications are single assertions which evaluate to true in every single state.

Number: $\dfrac{}{[q]\ n\ [q]}$	*Addition:* $\dfrac{[p]\ e_1\ [r] \quad [r]\ e_2\ [q]}{[p]\ e_1 + e_2\ [q]}$
Variable: $\dfrac{}{[q]\ x\ [q]}$	*Subtraction:* $\dfrac{[p]\ e_1\ [r] \quad [r]\ e_2\ [q]}{[p]\ e_1 - e_2\ [q]}$
Increment: $\dfrac{}{[q \lhd [(x+1)/x]]\ \ ++x\ [q]}$	*Multiplication:* $\dfrac{[p]\ e_1\ [r] \quad [r]\ e_2\ [q]}{[p]\ e_1 * e_2\ [q]}$

Table 6.3: Total Correctness of Numeric Expressions.

These are used to express side conditions of rules, some of which will eventually become verification conditions.

6.1.2 Numeric Expression Specification

$$[a_1]\ e\ [a_2]$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ e & : & \text{numeric expression} \\ a_2 & : & \text{postcondition} \end{array}$$

6.1.2.1 Semantics of Numeric Expression Specification

$$\begin{array}{rcl} [a_1]\ e\ [a_2] & = & (\forall s_1\ n\ s_2.\ A\ a_1\ s_1 \wedge E\ e\ s_1\ n\ s_2 \Rightarrow A\ a_2\ s_2) \wedge \\ & & (\forall s_1.\ A\ a_1\ s_1 \Rightarrow (\exists n\ s_2.\ E\ e\ s_1\ n\ s_2)) \end{array}$$

Null list:	*Cons:*
$\overline{[q]\ \langle\ \rangle\ [q]}$	$\dfrac{[p]\ e\ [r] \qquad [r]\ es\ [q]}{[p]\ CONS\ e\ es\ [q]}$

Table 6.4: Total Correctness of Expression Lists.

If the numeric expression e is executed, beginning in a state satisfying a_1, then the execution terminates in a state satisfying a_2. For this language, expressions are deterministic and always terminate.

Table 6.3 presents the rules of inference for individual constructors of numeric expressions in the Sunrise programming language. These are subsumed by the single rule in Table 6.2 for numeric expression preconditions, but are presented here for completeness.

The translation function VE maps a programming language numeric expression e into a corresponding assertion language numeric expression v, such that the value of v in the prior state, where a_1 is true, is the same as the value yielded by the execution of e.

6.1.3 Expression List Specification

$$[a_1]\ es\ [a_2]$$

a_1 : precondition
es : list of numeric expressions
a_2 : postcondition

6.1.3.1 Semantics of Expression List Specification

$$\begin{array}{rcl} [a_1]\ es\ [a_2] & = & (\forall s_1\ ns\ s_2.\ A\ a_1\ s_1 \wedge ES\ es\ s_1\ ns\ s_2 \Rightarrow A\ a_2\ s_2) \wedge \\ & & (\forall s_1.\ A\ a_1\ s_1 \Rightarrow (\exists ns\ s_2.\ ES\ es\ s_1\ ns\ s_2)) \end{array}$$

If the list of numeric expressions es is executed, beginning in a state satisfying a_1, then the execution terminates in a state satisfying a_2. For this language, expression lists are deterministic and always terminate.

Table 6.4 presents the rules of inference for individual constructors of lists of expressions in the Sunrise programming language. In this language, lists are delimited by angle brackets (so $\langle\,\rangle$ is the empty list), and a new element is added at the head of a list by $CONS$. These are subsumed by the single rule in Table 6.2 for expression list preconditions, but are presented here for completeness.

The translation function VES maps a programming language list of numeric expressions es into a corresponding assertion language list of numeric expressions vs, such that the value of vs in the prior state, where a_1 is true, is the same as the value yielded by the execution of es.

6.1.4 Boolean Expression Specification

$$[a_1]\ b\ [a_2]$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ b & : & \text{boolean expression} \\ a_2 & : & \text{postcondition} \end{array}$$

6.1.4.1 Semantics of Boolean Expression Specification

$$\begin{array}{rcl} [a_1]\ b\ [a_2] & = & (\forall s_1\ t\ s_2.\ A\ a_1\ s_1 \wedge B\ b\ s_1\ t\ s_2 \Rightarrow A\ a_2\ s_2) \wedge \\ & & (\forall s_1.\ A\ a_1\ s_1 \Rightarrow (\exists t\ s_2.\ B\ b\ s_1\ t\ s_2)) \end{array}$$

Numeric Equals:

$$\frac{[p]\ e_1\ [r] \quad [r]\ e_2\ [q]}{[p]\ e_1 = e_2\ [q]}$$

Conjunction:

$$\frac{[p]\ b_1\ [r] \quad [r]\ b_2\ [q]}{[p]\ b_1 \wedge b_2\ [q]}$$

Less Than:

$$\frac{[p]\ e_1\ [r] \quad [r]\ e_2\ [q]}{[p]\ e_1 < e_2\ [q]}$$

Disjunction:

$$\frac{[p]\ b_1\ [r] \quad [r]\ b_2\ [q]}{[p]\ b_1 \vee b_2\ [q]}$$

Lexicographic Less Than:

$$\frac{[p]\ es_1\ [r] \quad [r]\ es_2\ [q]}{[p]\ es_1 \ll es_2\ [q]}$$

Negation:

$$\frac{[p]\ b\ [q]}{[p] \sim b\ [q]}$$

Table 6.5: Total Correctness of Boolean Expressions.

If the boolean expression b is executed, beginning in a state satisfying a_1, then the execution terminates in a state satisfying a_2. For this language, boolean expressions are deterministic and always terminate.

Table 6.5 presents the rules of inference for individual constructors of boolean expressions in the Sunrise programming language. These are subsumed by the single rule in Table 6.2 for boolean expression preconditions, but are presented here for completeness.

The translation function AB maps a programming language boolean expression b into a corresponding assertion language numeric expression a, such that the value of a in the prior state, where a_1 is true, is the same as the value yielded by the execution of b.

6.2 Hoare Logic for Partial Correctness

In this section we present a Hoare logic for the partial correctness of commands.

6.2.1 Partial Correctness Specification

$$\{a_1\}\ c\ \{a_2\}\ /\rho$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ c & : & \text{command} \\ a_2 & : & \text{postcondition} \\ \rho & : & \text{procedure environment} \end{array}$$

6.2.1.1 Semantics of Partial Correctness Specification

$$\{a_1\}\ c\ \{a_2\}\ /\rho \ = \ (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge C\ c\ \rho\ s_1\ s_2 \Rightarrow A\ a_2\ s_2)$$

Skip:

$$\frac{}{\{q\}\ \textbf{skip}\ \{q\}\ /\rho}$$

Abort:

$$\frac{}{\{a\}\ \textbf{abort}\ \{q\}\ /\rho}$$

Assignment:

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

Sequence:

$$\frac{\{p\}\ c_1\ \{r\}\ /\rho, \quad \{r\}\ c_2\ \{q\}\ /\rho}{\{p\}\ c_1\ ;\ c_2\ \{q\}\ /\rho}$$

Conditional:

$$\frac{\begin{array}{c}\{r_1\}\ c_1\ \{q\}\ /\rho \\ \{r_2\}\ c_2\ \{q\}\ /\rho\end{array}}{\begin{array}{r}\{AB\ b => ab\_pre\ b\ r_1 \mid ab\_pre\ b\ r_2\} \\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi}\ \{q\}\ /\rho\end{array}}$$

Iteration:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho \\ WF_c\ (\textbf{assert}\ a\ \textbf{with}\ v < x \\ \qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ g\ \rho \\ \{p\}\ c\ \{a \wedge (v < x)\}\ /\rho \\ \{a \wedge (AB\ b) \wedge (v = x) \Rightarrow ab\_pre\ b\ p\} \\ \{a \wedge \sim(AB\ b) \Rightarrow ab\_pre\ b\ q\}\end{array}}{\begin{array}{l}\{a\}\ \textbf{assert}\ a\ \textbf{with}\ v < x \\ \qquad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od}\ \{q\}\ /\rho\end{array}}$$

Rule of Adaptation:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho, \quad WF_c\ c\ g\ \rho, \quad WF_{xs}\ x, \quad DL\ x \\ x_0 = logicals\ x, \quad x_0' = variants\ x_0\ (FV_a\ q) \\ FV_c\ c\ \rho \subseteq x, \quad FV_a\ pre \subseteq x, \quad FV_a\ post \subseteq (x \cup x_0) \\ \{x_0 = x \wedge pre\}\ c\ \{post\}/\rho\end{array}}{\{pre \wedge ((\forall x.\ (post \lhd [x_0'/x_0] \Rightarrow q)) \lhd [x/x_0'])\}\ c\ \{q\}/\rho}$$

Procedure Call:

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

Table 6.6: Hoare Logic for Partial Correctness.

Precondition Strengthening:

$$\frac{\{p \Rightarrow a\} \quad \{a\}\ c\ \{q\}\ /\rho}{\{p\}\ c\ \{q\}\ /\rho}$$

Postcondition Weakening:

$$\frac{\{p\}\ c\ \{a\}\ /\rho \quad \{a \Rightarrow q\}}{\{p\}\ c\ \{q\}\ /\rho}$$

False Precondition:

$$\frac{}{\{\mathbf{false}\}\ c\ \{q\}\ /\rho}$$

Table 6.7: General rules for Partial Correctness.

If the command c is executed, beginning in a state satisfying a_1, then if the execution terminates, the final state satisfies a_2. For this language, commands are deterministic, but may not terminate.

The procedure environment ρ is defined to be *well-formed for partial correctness* if for every procedure p, its body is partially correct with respect to the given precondition and postcondition:

$$\begin{array}{rcl} WF_{env\_partial}\ \rho & = & \forall p.\ \ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \mathbf{in} \\ & & \qquad \mathbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \mathbf{in} \\ & & \qquad \mathbf{let}\ x_0 = logicals\ x\ \mathbf{in} \\ & & \qquad \{x_0 = x\ \wedge\ pre\}\ c\ \{post\}\ /\rho \end{array}$$

6.2.2 Partial Correctness Rules

Consider the Hoare logic in Tables 6.6 and 6.7 for partial correctness. This is a traditional Hoare logic, except that we have added $/\rho$ at the end of each specification to indicate the ubiquitous procedure environment. This must be used to resolve the semantics of procedure call. However, the environment ρ

never changes during the execution of the program, and hence could be deleted from every specification, being understood in context.

The rules describing the partial correctness of the commands of the Sunrise programming language includes phrases that concern total correctness. For example, the iteration command includes a **with** $v < x$ phrase in the syntax, and the iteration rule uses antecedents that include $v < x$ and $v = x$. This mechanism applies to proofs of termination, not to proofs of partial correctness. Nevertheless, it is important to include this mechanism here because eventually we wish to prove versions of these rules for total correctness, which *will* need the extra mechanism. These rules will be ultimately proven using the following rule:

$$\frac{\begin{array}{c}\{p\}\ c\ \{q\}\ /\rho \\ [p]\ c \Downarrow /\rho\end{array}}{[p]\ c\ [q]\ /\rho}$$

where $[p]\ c \Downarrow /\rho$ denotes the termination of the command c. For this rule to apply, the shape of the partial and total correctness versions must agree.

The functions WF_α, for various α, denote *well-formedness* conditions, which will be described later in Part III. In brief, these are generally simple syntactic checks on variable names and limits on the free variables of program phrases, checks that the signatures of procedure definitions and their calls match, and the exclusion of aliasing. These checks could be performed once at compile time for a program. $WF_{env\_syntax}\ \rho$ checks that these well-formedness criteria are met by each procedure definition in ρ. $WF_{envp}\ \rho$ includes the criteria of $WF_{env\_syntax}\ \rho$, but goes beyond in also requiring a semantic criterion, that the body of each procedure is partially correct with respect to the precondition and postcondition specified in the procedure header. We establish $WF_{envp}\ \rho$ by what we call *semantic stages*, which will be described later in Part III. In addition to the well-formedness notation, we also use FV_α to denote the free variables of a construct, ampersand (&) to append two lists together, and SL to convert a list into a set. DL is a predicate on a list, which determines if all the elements of the list are distinct.

Of particular interest are the Rule of Adaptation and the Procedure Call Rule. All global variables and variable and value parameters are carefully and correctly handled. These rules are completely sound and trustworthy, having been proved as theorems.

6.3 Procedure Entrance Logic

The Procedure Entrance Logic is the second of the three newly invented logics of this dissertation. It is based on five new correctness specifications, which are the *entrance specification*, the *precondition entrance specification*, the *calls entrance specification*, the *path entrance specification*, and the *recursion entrance specification*. Each of these is a relation, defined using the other relations and the underlying structural operational semantics relations. The common thread linking all of these is the purpose of relating a state at the beginning of a computation with a state reached at the entrance of a procedure called during the computation. The style of these five specifications is similar to partial correctness, in that there is no guarantee of reaching the entrance of any procedure, only that if the appropriate entrance is reached, then the entrance condition specified is true. This is contrasted with the Termination Logic to be presented later, which has more the style of total correctness.

All of the rules listed for this entrance logic have been mechanically proven as theorems from the underlying structural operational semantics.

6.3.1 Entrance Specification

$$\{a_1\}\ c \to p\ \{a_2\}\ /\rho$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ c & : & \text{command} \\ p & : & \text{procedure name} \\ a_2 & : & \text{entrance condition} \\ \rho & : & \text{procedure environment} \end{array}$$

6.3.1.1 Semantics of Entrance Specification

$$\{a_1\}\ c \to p\ \{a_2\}\ \rho \ = \ (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge C\_calls\ c\ \rho\ s_1\ p\ s_2 \Rightarrow A\ a_2\ s_2)$$

If command c is executed, beginning in a state satisfying a_1, then if at any point within c procedure p is called, then at the entry of p, (just before the body of p is executed,) a_2 is satisfied. This refers only to the first level of calls from c, to those that issue directly from a syntactically contained procedure call command within c. It does not refer to calls of p that may occur from the body of p, or of other procedures that c may call indirectly during the execution of c.

No statement is made here about conditions that may hold at the end of the execution of c.

Note that a particular command c may contain several calls of p, each of which might be responsible for entering p. Also, if c contains a loop, even a single call of p may generate multiple states at the entrance of p. Thus this is a relation, where for a single command and starting state, there may be many entrance states for

Precondition Strengthening:

$$\frac{\{a_0 \Rightarrow a_1\} \quad \{a_1\}\ c \to p\ \{a_2\}\ /\rho}{\{a_0\}\ c \to p\ \{a_2\}\ /\rho}$$

Entrance Condition Weakening:

$$\frac{\{a_1\}\ c \to p\ \{a_2\}\ /\rho \quad \{a_2 \Rightarrow a_3\}}{\{a_1\}\ c \to p\ \{a_3\}\ /\rho}$$

Entrance Condition Conjunction:

$$\frac{\{a_1\}\ c \to p\ \{a_2\}\ /\rho \quad \{a_1\}\ c \to p\ \{a_3\}\ /\rho}{\{a_1\}\ c \to p\ \{a_2 \wedge a_3\}\ /\rho}$$

False Precondition:

$$\frac{}{\{\mathbf{false}\}\ c \to p\ \{q\}\ /\rho}$$

Skip:

$$\frac{}{\{a\}\ \mathbf{skip} \to p\ \{q\}\ /\rho}$$

Abort:

$$\frac{}{\{a\}\ \mathbf{abort} \to p\ \{q\}\ /\rho}$$

Assignment:

$$\frac{}{\{a\}\ x := e \to p\ \{q\}\ /\rho}$$

Sequence:

$$\frac{\{a_1\}\ c_1 \to p\ \{q\}\ /\rho \quad \{a_1\}\ c_1\ \{a_2\}\ /\rho \quad \{a_2\}\ c_2 \to p\ \{q\}\ /\rho}{\{a_1\}\ c_1\ ;\ c_2 \to p\ \{q\}\ /\rho}$$

Conditional:

$$\frac{\{a_1\}\ c_1 \to p\ \{q\}\ /\rho \quad \{a_2\}\ c_2 \to p\ \{q\}\ /\rho}{\{AB\ b => ab\_pre\ b\ a_1 \mid ab\_pre\ b\ a_2\}\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \to p\ \{q\}\ /\rho}$$

Iteration:

$$\frac{\begin{array}{c} WF_{env\_syntax}\ \rho \\ WF_c\ (\mathbf{assert}\ a\ \mathbf{with}\ v < x\ \ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ g\ \rho \\ x \notin FV_a\ q \\ [a \wedge (AB\ b) \wedge (v = x)]\ b\ [a_0] \\ \{a_0\}\ c \to p\ \{q\}\ /\rho \\ \{a_0\}\ c\ \{a \wedge (v < x)\}\ /\rho \end{array}}{\{a\}\ \mathbf{assert}\ a\ \mathbf{with}\ v < x\ \ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od} \to p\ \{q\}\ /\rho}$$

Procedure Call:

$$\frac{p_1 \neq p}{\{a\}\ \mathbf{call}\ p_1(xs; es) \to p\ \{q\}\ /\rho}$$

$$\frac{\begin{array}{c} WF_{env\_syntax}\ \rho \\ WF_c\ (\mathbf{call}\ p(xs; es))\ g\ \rho \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \end{array}}{\begin{array}{c}\{(q \lhd [xs\ \&\ vals'/vars\ \&\ vals]) \lhd [vals' := es]\} \\ \mathbf{call}\ p(xs; es) \to p\ \{q\}\ /\rho\end{array}}$$

Table 6.8: Entrance Logic.

which the entrance condition is to hold.

Table 6.8 presents an *axiomatic entrance semantics* for the Sunrise programming languge.

6.3.1.2 Example of Entrance Specification

As an example, consider the progress claimed for calls from procedure *odd* to procedure *even* in the odd/even program presented in Table 6.1. In the heading for procedure *odd*, the phrase **calls** *even* **with** $n < \hat{n}$ indicates that the value of the n argument to *even* must be strictly less than the value of n at the head of the body of *odd*.

First, by the Procedure Call rule of Table 6.8 applied to the call $even(a; n-1)$ within the body of procedure *odd*, we have

$$\{((n < \hat{n}) \lhd [a, n/a, n]) \lhd [n := n-1]\} \ \mathbf{call} \ even(a; n-1) \rightarrow even \ \{n < \hat{n}\} \ /\rho$$

The substitutions evaluate as

$$\begin{aligned} ((n < \hat{n}) \lhd [a, n/a, n]) \lhd [n := n-1] &= (n < \hat{n}) \lhd [n := n-1] \\ &= (n-1) < \hat{n} \end{aligned}$$

Then the call progress claim is proven as follows.

1.	$\{(n-1) < \widehat{n}\}\ even(a; n-1) \rightarrow even\ \{n < \widehat{n}\}\ /\rho$	Procedure Call Rule (2nd)
2.	$\{\mathbf{true}\}\ odd(a; n-2) \rightarrow even\ \{n < \widehat{n}\}\ /\rho$	Procedure Call Rule (1st)
3.	$\{n = 1 => (n-1) < \widehat{n} \mid \mathbf{true}\}$ **if** $n = 1$ **then** $even(a; n-1)$ **else** $odd(a; n-2)$ **fi** $\rightarrow even\ \{n < \widehat{n}\}\ /\rho$	1, 2, Conditional Rule
4.	$\{\mathbf{true}\}\ a := 0 \rightarrow even\ \{n < \widehat{n}\}\ /\rho$	Assignment Rule
5.	$\{n = 0 => \mathbf{true}$ $\mid (n = 1 => (n-1) < \widehat{n} \mid \mathbf{true})\}$ **if** $n = 0$ **then** $a := 0$ **else if** $n = 1$ **then** $even(a; n-1)$ **else** $odd(a; n-2)$ **fi** **fi** $\rightarrow even\ \{n < \widehat{n}\}\ /\rho$	4, 3, Conditional Rule
6.	$\{\widehat{n} = n \Rightarrow\ (n = 0 => \mathbf{true}$ $\mid (n = 1 => (n-1) < \widehat{n} \mid \mathbf{true})\ )\}$	Tautology
7.	$\{\widehat{n} = n\}$ **if** $n = 0$ **then** $a := 0$ **else if** $n = 1$ **then** $even(a; n-1)$ **else** $odd(a; n-2)$ **fi** **fi** $\rightarrow even\ \{n < \widehat{n}\}\ /\rho$	6, 5, Precondition Strengthening

A similar pattern of reasoning could be followed to prove the clause

$$\mathbf{calls}\ odd\ \mathbf{with}\ n < \widehat{n}$$

in the heading for procedure *odd*, and the other such clauses in the heading for *even*.

6.3.2 Precondition Entrance Specification

$$\{a\}\ c \rightarrow \textbf{pre}\ /\rho$$

$$\begin{array}{rcl} a & : & \text{precondition} \\ c & : & \text{command} \\ \rho & : & \text{procedure environment} \end{array}$$

6.3.2.1 Semantics of Precondition Entrance Specification

$$\begin{array}{rcl} \{a\}\ c \rightarrow \textbf{pre}\ /\rho & = & \forall p.\ \ \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c'\rangle = \rho\ p \\ & & \textbf{in}\ \{a\}\ c \rightarrow p\ \{pre\}\ /\rho \end{array}$$

If command c is executed, beginning in a state satisfying a, then if at any point within c a call is made to any procedure, say p, then at the entry of p, the declared precondition of p is satisfied.

This specification is used to prove that the preconditions which are declared for each procedure in its header are achieved at the point of each call of those procedures within the command c. Eventually, this will be used to prove the maintenance of preconditions, that for each procedure, if it is entered with its precondition true, then for every procedure it calls, their preconditions are true at their entry. This will then extend to the maintenance of preconditions over deep chains of calls.

The procedure environment ρ is defined to be *well-formed for preconditions* if for every procedure p, its body maintains all procedures' preconditions:

$$\begin{array}{rcl} WF_{env\_pre}\ \rho & = & \forall p.\ \ \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p \\ & & \textbf{in}\ \{pre\}\ c \rightarrow \textbf{pre}\ /\rho \end{array}$$

Proving that the environment is well-formed for preconditions is one of the necessary steps to prove programs totally correct.

6.3.3 Calls Entrance Specification

$$\{a\}\ c \rightarrow calls\ /\rho$$

a	:	precondition
c	:	command
$calls$	:	calls progress environment
ρ	:	procedure environment

6.3.3.1 Semantics of Calls Entrance Specification

$$\{a\}\ c \rightarrow calls\ /\rho\ \ =\ \ \forall p.\ \{a\}\ c \rightarrow p\ \{calls\ p\}\ /\rho$$

calls is a collection of progress expressions, as declared in the **calls** ... **with** specifications for a procedure in its header. It is represented as a function, from the names of procedures being called to the progress expression specified.

If command c is executed, beginning in a state satisfying a, then if at any point within c a call is made to any procedure, say p, then at the entry of p, $(calls\ p)$ is satisfied.

This specification is used to prove that the progress expressions which are declared in the **calls** ... **with** specifications for each procedure in its header are achieved at the point of each call of those procedures within the command c.

The procedure environment ρ is defined to be *well-formed for calls progress* if for every procedure p, its body establishes the truth of its *calls* progress expressions at the point of each call:

$$\begin{array}{rcll} WF_{env\_calls}\ \rho & = & \forall p. & \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \textbf{in} \\ & & & \textbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \textbf{in} \\ & & & \textbf{let}\ x_0 = logicals\ x\ \textbf{in} \\ & & & \{x_0 = x \wedge pre\}\ c \rightarrow calls\ /\rho \end{array}$$

Proving that the environment is well-formed for calls progress is one of the necessary steps to prove programs totally correct.

Eventually, $WF_{env\_calls}\ \rho$ will be used to prove the **recurses with** specifications, that for each procedure, if it is entered with its recursion expression equal to a certain value, then for every possible recursive entry of that procedure, the value of the recursion expression is strictly less than before. This will then help prove the termination of procedures.

Up to this point, the entrance specifications have been based on a command over which the progress was measured. For the last two entrance specifications in this Procedure Entrance Logic, they will be based on progress from one entrance of a procedure to another.

6.3.4 Path Entrance Specification

$$\{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ /\rho$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ p_1 & : & \text{starting procedure name} \\ ps & : & \text{path (list of procedure names)} \\ p_2 & : & \text{destination procedure name} \\ a_2 & : & \text{entrance condition} \\ \rho & : & \text{procedure environment} \end{array}$$

6.3.4.1 Semantics of Path Entrance Specification

$$\begin{array}{l} \{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ \rho\ = \\ \qquad (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge M\_calls\ p_1\ s_1\ ps\ p_2\ s_2\ \rho \Rightarrow A\ a_2\ s_2) \end{array}$$

If execution begins at the entry of p_1 in a state satisfying a_1, and if in the execution of the body of p_1, procedure calls are made successively deeper to the procedures listed in the (possibly empty) path ps, and finally a call is made to

Single Call (Empty Path):

$$\frac{\rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \quad \{a_1\}\ c \rightarrow p_2\ \{a_2\}/\rho}{\{a_1\}\ p_1 \text{ --- } \langle\ \rangle \rightarrow p_2\ \{a_2\}/\rho}$$

Transitivity:

$$\frac{\{a_1\}\ p_1 \text{ --- } ps_1 \rightarrow p_2\ \{a_2\}/\rho \quad \{a_2\}\ p_2 \text{ --- } ps_2 \rightarrow p_3\ \{a_3\}/\rho}{\{a_1\}\ p_1 \text{ --- } (ps_1\ \&\ (CONS\ p_2\ ps_2)) \rightarrow p_3\ \{a_3\}/\rho}$$

Table 6.9: Path Entrance Logic.

the procedure p_2, then at that entry of p_2, a_2 is satisfied.

The path entrance specification is defined based on the underlying operational semantics. However, it could have been defined by rule induction on the rules in Table 6.9. Instead, these rules have been proven as theorems, as have those in Table 6.10.

Once the environment ρ is proven to be well-formed for preconditions, the following rule, proven as a theorem, applies for proving the truth of preconditions across procedure calls.

$$\frac{WF_{env\_pre}\ \rho \quad \rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \quad \rho\ p_2 = \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle}{\{pre\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{pre'\}\ /\rho}$$

Precondition Strengthening:

$$\frac{\{a_0 \Rightarrow a_1\} \qquad \{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ /\rho}{\{a_0\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ /\rho}$$

Entrance Condition Conjunction:

$$\frac{\{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ /\rho \qquad \{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_3\}\ /\rho}{\{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2 \wedge a_3\}\ /\rho}$$

Entrance Condition Weakening:

$$\frac{\{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_2\}\ /\rho \qquad \{a_2 \Rightarrow a_3\}}{\{a_1\}\ p_1 - ps \rightarrow p_2\ \{a_3\}\ /\rho}$$

False Precondition:

$$\frac{}{\{\mathbf{false}\}\ p_1 - ps \rightarrow p_2\ \{q\}\ /\rho}$$

Table 6.10: Additional Path Entrance Rules.

6.3.4.2 Call Progress Function

Just as the *ab_pre* function can compute the appropriate precondition to establish a given postcondition as true after executing a boolean expression, the *call_progress* function can compute the appropriate precondition when starting execution from the entrance of one procedure to establish a given entrance condition for another procedure as true. It is defined in Table 6.11.

Once the environment ρ is proven to be well-formed for calls progress, the following rule, proven as a theorem, applies for proving the effect of the *call_progress* function across a single procedure call.

Call Progress Rule:

$$\frac{\begin{array}{c} WF_{env\_syntax}\ \rho, \quad WF_{env\_calls}\ \rho \\ \rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ \rho\ p_2 = \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle \\ y = vars'\ \&\ vals'\ \&\ glbs' \\ FV_a\ q \subseteq SL(y\ \&\ logicals\ z) \end{array}}{\{pre \wedge call\_progress\ p_1\ p_2\ q\ \rho\}\ p_1 - \langle\ \rangle \rightarrow p_2\ \{q\}\ /\rho}$$

$$
\begin{array}{l}
\mathit{call\_progress}\ p_1\ p_2\ q\ \rho\ \ = \\
\qquad \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p_1\ \mathbf{in} \\
\qquad \mathbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \mathbf{in} \\
\qquad \mathbf{let}\ x_0 = logicals\ x\ \mathbf{in} \\
\qquad \mathbf{let}\ x_0' = variants\ x_0\ (FV_a\ q)\ \mathbf{in} \\
\qquad \mathbf{let}\ \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle = \rho\ p_2\ \mathbf{in} \\
\qquad \mathbf{let}\ y = vars'\ \&\ vals'\ \&\ glbs'\ \mathbf{in} \\
\qquad \mathbf{let}\ a = calls\ p_2\ \mathbf{in} \\
\qquad (\ a = \mathbf{false} \Rightarrow \mathbf{true} \\
\qquad\qquad\qquad \mid (\forall y.\ (a \lhd [x_0'/x_0]) \Rightarrow q) \lhd [x/x_0']\ )
\end{array}
$$

Table 6.11: Call Progress Function.

6.3.4.3 Example of Call Progress Specification

As an example, consider the progress of calls from procedure *odd* to procedure *even* in the odd/even program presented in Table 6.1. We previously proved the correctness of the claim in the heading for procedure *odd* that **calls** *even* **with** $n < \hat{n}$, that the value of the n argument to *even* must be strictly less than the value of n at the head of the body of *odd*.

Then by the Call Progress Rule given above, we have

$$\{\mathbf{true} \wedge call\_progress\ odd\ even\ (n < \hat{n})\ \rho\}\ odd - \langle\ \rangle \rightarrow even\ \{n < \hat{n}\}\ /\rho$$

The invocation of *call_progress* evaluates as

$$
\begin{array}{rl}
\multicolumn{2}{l}{call\_progress\ odd\ even\ (n < \hat{n})\ \rho} \\
= & (\forall a, n.\ ((n < \hat{n}) \lhd [\hat{a}, \widehat{n_1}/\hat{a}, \hat{n}]) \Rightarrow (n < \hat{n})) \lhd [a, n/\hat{a}, \widehat{n_1}] \\
= & (\forall a, n.\ (n < \widehat{n_1}) \Rightarrow (n < \hat{n})) \lhd [a, n/\hat{a}, \widehat{n_1}] \\
= & (\forall a_1, n_1.\ (n_1 < \widehat{n_1}) \Rightarrow (n_1 < \hat{n})) \lhd [a, n/\hat{a}, \widehat{n_1}] \\
= & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})
\end{array}
$$

Thus we have proven

$$\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})\}\ odd — \langle\rangle \rightarrow even\ \{n < \hat{n}\}\ /\rho$$

A similar pattern of reasoning could be followed to prove the following:

$$\begin{aligned}&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})\}\\ &\qquad\qquad even — \langle\rangle \rightarrow even\ \{n < \hat{n}\}\ /\rho\end{aligned}$$

$$\begin{aligned}&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \hat{n}))\}\\ &\qquad\qquad odd — \langle\rangle \rightarrow odd\ \{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})\}\ /\rho\end{aligned}$$

$$\begin{aligned}&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \hat{n}))\}\\ &\qquad\qquad even — \langle\rangle \rightarrow odd\ \{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n})\}\ /\rho\end{aligned}$$

6.3.4.4 Call Path Progress Function

Just as the *call_progress* function can compute the appropriate precondition across a single procedure call, the *call_path_progress* function can compute the appropriate precondition when starting execution from the entrance of one procedure to establish a given entrance condition at the end of a path of procedure calls. It is defined in Table 6.12.

$$\begin{aligned}&call\_path\_progress\ p_1\ \langle\ \rangle\ p_2\ q\ \rho\ =\\ &\qquad call\_progress\ p_1\ p_2\ q\ \rho\\ \\ &call\_path\_progress\ p_1\ (CONS\ p\ ps)\ p_2\ q\ \rho\ =\\ &\qquad call\_progress\ p_1\ p\ (call\_path\_progress\ p\ ps\ p_2\ q\ \rho)\ \rho\end{aligned}$$

Table 6.12: Call Path Progress Function.

Once the environment ρ is proven to be well-formed for preconditions *and* for calls progress, the following rule, proven as a theorem, applies for proving the effect of the *call_path_progress* function across a path of procedure calls.

Call Path Progress Rule:

$$\frac{\begin{array}{c} WF_{env\_syntax}\ \rho, \quad WF_{env\_pre}\ \rho, \quad WF_{env\_calls}\ \rho \\ \rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ \rho\ p_2 = \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle \\ y = vars'\ \&\ vals'\ \&\ glbs' \\ FV_a\ q \subseteq SL(y\ \&\ logicals\ z) \end{array}}{\{pre \wedge call\_path\_progress\ p_1\ ps\ p_2\ q\ \rho\}\ p_1 \;—\; ps \rightarrow p_2\ \{q\}\ /\rho}$$

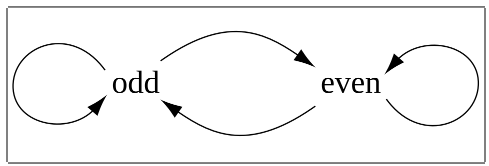

Figure 6.2: Procedure Call Graph for Odd/Even Example.

6.3.4.5 Example of Call Path Progress Specification

As an example, consider the progress of paths of procedure calls that involve the procedure *even* in the odd/even program presented in Table 6.1. Examining the procedure call graph in Figure 6.2, we can observe several cycles that include the *even* node. Let us assume the correctness of the call progress parts of the headers of the procedures as declared, that is, that every **calls** ... **with** clause has been verified to be true.

Then consider the path $odd \rightarrow odd \rightarrow even$. By the Call Path Progress Rule given above, we have

$$\begin{array}{c} \{\mathbf{true} \wedge call\_path\_progress\ odd\ \langle odd\rangle\ even\ (n < \widehat{n})\ \rho\} \\ odd \;—\; \langle odd\rangle \rightarrow even\ \{n < \widehat{n}\}\ /\rho \end{array}$$

We previously evaluated $call\_progress\ odd\ even\ (n < \widehat{n})\ \rho$ as

$$call\_progress\ odd\ even\ (n < \widehat{n})\ \rho$$

$$= \quad \forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n})$$

Using this, we can evaluate the invocation of *call_path_progress* as

$$call\_path\_progress\ odd\ \langle odd\rangle\ even\ (n < \widehat{n})\ \rho$$

$$\begin{aligned}
&= call_progress\ odd\ odd\ (call_path_progress\ odd\ \langle\rangle\ even\ (n < \widehat{n})\ \rho)\ \rho \\
&= call_progress\ odd\ odd\ (call_progress\ odd\ even\ (n < \widehat{n})\ \rho)\ \rho \\
&= call_progress\ odd\ odd\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n}))\ \rho \\
&= (\forall a, n.\ ((n < \widehat{n}) \lhd [\widehat{a}, \widehat{n_1}/\widehat{a}, \widehat{n}]) \Rightarrow \\
&\qquad\qquad (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n}))) \lhd [a, n/\widehat{a}, \widehat{n_1}] \\
&= (\forall a, n.\ (n < \widehat{n_1}) \Rightarrow (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n}))) \lhd [a, n/\widehat{a}, \widehat{n_1}] \\
&= (\forall a_1, n_1.\ (n_1 < \widehat{n_1}) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n}))) \lhd [a, n/\widehat{a}, \widehat{n_1}] \\
&= \forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n}))
\end{aligned}$$

Thus we have proven

$$\begin{aligned}
&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n}))\} \\
&\qquad\qquad odd \text{ --- } \langle odd\rangle \rightarrow even\ \{n < \widehat{n}\}\ /\rho
\end{aligned}$$

Similar patterns of reasoning could be followed to prove the following:

$$\begin{aligned}
&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n}))\} \\
&\qquad\qquad even \text{ --- } \langle odd\rangle \rightarrow even\ \{n < \widehat{n}\}\ /\rho
\end{aligned}$$

$$\begin{aligned}
&\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n})\} \\
&\qquad\qquad even \text{ --- } \langle\rangle \rightarrow even\ \{n < \widehat{n}\}\ /\rho
\end{aligned}$$

6.3.5 Recursive Entrance Specification

$$\{a_1\}\ p \hookleftarrow \{a_2\}\ /\rho$$

a_1	:	precondition
p	:	procedure name
a_2	:	recursive entrance condition
ρ	:	procedure environment

6.3.5.1 Semantics of Recursive Entrance Specification

$$\{a_1\}\ p \hookleftarrow \{a_2\}\ /\rho \ = \ \forall ps.\ \{a_1\}\ p \longrightarrow ps \rightarrow p\ \{a_2\}\ /\rho$$

If execution begins at the entry of p in a state satisfying a_1, and if in the execution of the body of p, a (possibly deeply nested) recursive call is made to the procedure p, then at that recursive entry of p, a_2 is satisfied.

This specification is used to prove that procedures terminate, by a well-founded induction on the value of the recursive expression of each procedure.

When a procedure is declared, the recursion expression which is specified may be of two forms. It may be simply **false**, which signifies that the procedure is not recursive. Else, it may be of the form $v < x$, where v is a numeric assertion language expression whose free variables consist only of the parameters and globals of the procedure, and where x is a logical variable. v is the important part here; such a recursion expression signifies that v strictly decreases between recursive calls. This then is used to prove termination.

Based on these two cases, there are two initial expressions whose truth guarantees the achievement of the recursion expression:

$$\begin{aligned} \mathit{induct\_pre}\ \mathbf{false} &= \mathbf{true} \\ \mathit{induct\_pre}\ (v < x) &= (v = x) \end{aligned}$$

The procedure environment ρ is defined to be *well-formed for recursion* if for every procedure p, it establishes the truth of its recursion expression for every recursive call:

$$\begin{aligned} WF_{env\_rec}\ \rho \ = \ \forall p.\ & \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p \\ & \mathbf{in}\ \{pre \wedge \mathit{induct\_pre}\ rec\}\ p \hookleftarrow \{rec\}\ /\rho \end{aligned}$$

Proving that the environment is well-formed for recursion is one of the necessary steps to prove programs totally correct.

Eventually, $WF_{env\_rec}\ \rho$ will be used to prove the termination of each procedure. This will then help prove the termination of all commands, and the total correctness of all commands.

6.4 Termination Logic

The Termination Logic is the third of the three newly invented logics of this dissertation. It is based on three new correctness specifications, which are the *command conditional termination specification*, the *procedure conditional termination specification*, and the *command termination specification.* Each of these is a relation, defined using the other relations and the underlying structural operational semantics relations. The style of these three specifications is similar to total correctness, in that the specification simply guarantees that the computation terminates, without any claim about the terminal state itself. This is contrasted with the Procedure Entrance Logic presented earlier, which has more the style of partial correctness.

All of the rules listed for this termination logic have been mechanically proven as theorems from the underlying structural operational semantics.

The two "conditional termination" specifications involve a conditional quality, where the termination described is conditioned on the termination of all immediate calls issuing from the computation at the top level. In other words, if we are given that all procedure calls terminate which are made at the top level of the command or procedure body concerned, then the command or procedure body itself terminates.

This is the important issue to consider at this point, because after verifying the partial correctness axiomatic semantics given in section 6.2, it is then possible to prove the termination, and hence the total correctness, of every command in the Sunrise programming language *except* for procedure calls. For example, given the termination of every procedure call issuing from the body of a while loop, we could prove without further mechanism the total correctness of the while loop. The remaining kind of termination which is not yet covered is infinite recursive descent, where a cycle of procedures call each other in an ever descending sequence of procedure calls, none of which ever return.

The purpose of the Procedure Entrance Logic given in section 6.3 is to provide a means to prove the termination of procedure calls, by showing that a certain kind of progress is achieved between recursive entrances of the same procedure. The purpose of the Termination Logic of this section is to take that means, and prove the termination of commands and procedures. But we begin by proving *conditional termination*, by which we mean a kind of conditional termination depending on the termination of all immediate calls.

6.4.1 Command Conditional Termination Specification

$$[a]\ c \downarrow / \rho$$

a	:	precondition
c	:	command
ρ	:	procedure environment

6.4.1.1 Semantics of Command Conditional Termination Specification

$$[a]\ c \downarrow /\rho \ = \ (\forall s_1.\ A\ a\ s_1 \wedge C\_calls\_terminate\ c\ \rho\ s_1 \Rightarrow (\exists s_2.\ C\ c\ \rho\ s_1\ s_2))$$

If command c is executed, beginning in a state satisfying a, and if all calls issuing immediately from c terminate, then c terminates. This refers only to the first level of calls from c, to those that issue directly from a syntactically contained procedure call command within c. It does not refer to calls of p that may occur from the body of p, or of other procedures that c may call indirectly during the execution of c.

No statement is made here about conditions that may hold at the end of the execution of c.

Table 6.13 presents an *axiomatic termination semantics* for the Sunrise programming languge.

The procedure environment ρ is defined to be *well-formed for conditional termination* if for every procedure p, the body of p terminates given the termination of all immediate calls from the body:

$$\begin{array}{lcl} WF_{env\_term}\ \rho & = & \forall p.\ \ \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p \\ & & \textbf{in}\ [pre]\ c \downarrow /\rho \end{array}$$

Proving that the environment is well-formed for conditional termination is one of the necessary steps to prove programs totally correct.

Eventually, $WF_{env\_term}\ \rho$ will be used to prove the termination of each procedure, not conditionally on its immediate calls, but absolutely. This will then help prove the termination of all commands, and the total correctness of all commands.

Precondition Strengthening:

$$\frac{\{a_0 \Rightarrow a_1\} \quad [a_1]\ c \downarrow /\rho}{[a_0]\ c \downarrow /\rho}$$

False Precondition:

$$\frac{}{[\textbf{false}]\ c \downarrow /\rho}$$

Skip:

$$\frac{}{[a]\ \textbf{skip} \downarrow /\rho}$$

Abort:

$$\frac{}{[\textbf{false}]\ \textbf{abort} \downarrow /\rho}$$

Assignment:

$$\frac{}{[a]\ x := e \downarrow /\rho}$$

Sequence:

$$\frac{[a_1]\ c_1 \downarrow /\rho \quad \{a_1\}\ c_1\ \{a_2\}\ /\rho \quad [a_2]\ c_2 \downarrow /\rho}{[a_1]\ c_1\ ;\ c_2 \downarrow /\rho}$$

Conditional:

$$\frac{[a_1]\ c_1 \downarrow /\rho \quad [a_2]\ c_2 \downarrow /\rho}{[AB\ b => ab\_pre\ b\ a_1 \mid ab\_pre\ b\ a_2]\ \ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi} \downarrow /\rho}$$

Iteration:

$$\frac{\begin{array}{c} WF_{env\_syntax}\ \rho \\ WF_c\ (\textbf{assert}\ a\ \textbf{with}\ v < x\ \ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ g\ \rho \\ [a \wedge (AB\ b) \wedge (v = x)]\ b\ [a_0] \\ [a_0]\ c \downarrow /\rho \\ \{a_0\}\ c\ \{a \wedge (v < x)\}\ /\rho \end{array}}{[a]\ \textbf{assert}\ a\ \textbf{with}\ v < x\ \ \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od} \downarrow /\rho}$$

Procedure Call:

$$\frac{WF_{env\_syntax}\ \rho \quad WF_c\ (\textbf{call}\ p(xs; es))\ g\ \rho}{[a]\ \textbf{call}\ p(xs; es) \downarrow /\rho}$$

Table 6.13: Command Conditional Termination Logic.

6.4.2 Procedure Conditional Termination Specification

$$p \downarrow / \rho$$

$$\begin{array}{lcl} p & : & \text{procedure name} \\ \rho & : & \text{procedure environment} \end{array}$$

6.4.2.1 Semantics of Procedure Conditional Termination Specification

$$\begin{array}{rl} p \downarrow / \rho = & \textbf{let } \langle vars, vals, glbs, pre, post, calls, rec, c \rangle = \rho\ p \textbf{ in} \\ & [pre]\ c \downarrow / \rho \end{array}$$

If procedure p is entered in a state which satisfies the precondition of p, and if all calls issuing immediately from the body of p terminate, then p terminates.

This specification extends command conditional termination specifications to the bodies of procedures, and fixes the precondition to be the declared precondition of the procedure involved.

Once the environment ρ is proven to be well-formed for conditional termination, the following rule, proven as a theorem, says that all procedures conditionally terminate.

$$\frac{\begin{array}{c} WF_{env\_term}\ \rho \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c \rangle \end{array}}{p \downarrow / \rho}$$

6.4.3 Command Termination Specification

$$[a]\ c \Downarrow / \rho$$

$$\begin{array}{lcl} a & : & \text{precondition} \\ c & : & \text{command} \\ \rho & : & \text{procedure environment} \end{array}$$

Precondition Strengthening:	*False Precondition:*
$$\frac{\{p \Rightarrow q\} \quad [q]\ c \Downarrow /\rho}{[p]\ c \Downarrow /\rho}$$	$$\frac{}{[\mathbf{false}]\ c \Downarrow /\rho}$$

Table 6.14: General rules for Command Termination.

6.4.3.1 Semantics of Command Termination Specification

$$[a]\ c \Downarrow /\rho \ = \ (\forall s_1.\ A\ a\ s_1 \Rightarrow (\exists s_2.\ C\ c\ \rho\ s_1\ s_2))$$

If command c is executed, beginning in a state satisfying a, then c terminates.

No statement is made here about conditions that may hold at the end of the execution of c.

Tables 6.14 and 6.15 present an *axiomatic termination semantics* for the Sunrise programming languge.

The procedure environment ρ is defined to be *well-formed for termination* if for every procedure p, its body terminates with respect to the given precondition:

$$\begin{array}{lcl} WF_{env\_total}\ \rho & = & \forall p.\ \ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \mathbf{in} \\ & & \qquad \mathbf{let}\ x = vars \,\&\, vals \,\&\, glbs\ \mathbf{in} \\ & & \qquad \mathbf{let}\ x_0 = logicals\ x\ \mathbf{in} \\ & & \qquad [x_0 = x\ \wedge\ pre]\ c \Downarrow /\rho \end{array}$$

Skip:

$$\frac{}{[q]\ \textbf{skip} \Downarrow /\rho}$$

Abort:

$$\frac{}{[\textbf{false}]\ \textbf{abort} \Downarrow /\rho}$$

Assignment:

$$\frac{}{[a]\ x := e \Downarrow /\rho}$$

Sequence:

$$\frac{\begin{array}{c}[p]\ c_1 \Downarrow /\rho \\ \{p\}\ c_1\ \{q\}/\rho \\ [q]\ c_2 \Downarrow /\rho\end{array}}{[p]\ c_1\ ;\ c_2 \Downarrow /\rho}$$

Conditional:

$$\frac{\begin{array}{c}[r_1]\ c_1 \Downarrow /\rho \\ [r_2]\ c_2 \Downarrow /\rho\end{array}}{\begin{array}{c}[AB\ b => ab\_pre\ b\ r_1 \mid ab\_pre\ b\ r_2] \\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi} \Downarrow /\rho\end{array}}$$

Iteration:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho \\ WF_c\ (\textbf{assert}\ a\ \textbf{with}\ v < x \\ \quad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ g\ \rho \\ \{p\}\ c\ \{a \wedge (v < x)\}/\rho \\ [p]\ c \Downarrow /\rho \\ \{a \wedge (AB\ b) \wedge (v = x) \Rightarrow ab\_pre\ b\ p\} \\ \{a \wedge \sim(AB\ b) \Rightarrow ab\_pre\ b\ q\}\end{array}}{\begin{array}{c}[a]\ \textbf{assert}\ a\ \textbf{with}\ v < x \\ \quad \textbf{while}\ b\ \textbf{do}\ c\ \textbf{od} \Downarrow /\rho\end{array}}$$

Rule of Adaptation:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho, \quad WF_c\ c\ g\ \rho, \quad WF_{xs}\ x, \quad DL\ x \\ x_0 = logicals\ x, \quad x_0' = variants\ x_0\ (FV_a\ q) \\ FV_c\ c\ \rho \subseteq x, \quad FV_a\ pre \subseteq x, \quad FV_a\ post \subseteq (x \cup x_0) \\ [x_0 = x \wedge pre]\ c \Downarrow /\rho\end{array}}{[pre \wedge ((\forall x.\ (post \lhd [x_0'/x_0] \Rightarrow q)) \lhd [x/x_0'])]\ c \Downarrow /\rho}$$

Procedure Call:

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

Table 6.15: Hoare Logic for Command Termination.

Precondition Strengthening:

$$\frac{\begin{array}{c}\{p \Rightarrow a\} \\ [a]\ c\ [q]\ /\rho\end{array}}{[p]\ c\ [q]\ /\rho}$$

Postcondition Weakening:

$$\frac{\begin{array}{c}[p]\ c\ [a]\ /\rho \\ \{a \Rightarrow q\}\end{array}}{[p]\ c\ [q]\ /\rho}$$

False Precondition:

$$\frac{}{[\textbf{false}]\ c\ [q]\ /\rho}$$

Table 6.16: General rules for Total Correctness.

6.5 Hoare Logic for Total Correctness

6.5.1 Total Correctness Specification

$$[a_1]\ c\ [a_2]\ /\rho$$

$$\begin{array}{lcl} a_1 & : & \text{precondition} \\ c & : & \text{command} \\ a_2 & : & \text{postcondition} \\ \rho & : & \text{procedure environment} \end{array}$$

6.5.1.1 Semantics of Total Correctness Specification

$$\begin{array}{rcl} [a_1]\ c\ [a_2]\ /\rho & = & (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge C\ c\ \rho\ s_1\ s_2 \Rightarrow A\ a_2\ s_2) \wedge \\ & & (\forall s_1.\ A\ a_1\ s_1 \Rightarrow (\exists s_2.\ C\ c\ \rho\ s_1\ s_2)) \end{array}$$

If the command c is executed, beginning in a state satisfying a_1, then the execution terminates in a state satisfying a_2.

Consider the Hoare logic in Tables 6.16 and 6.17 for total correctness. This is a traditional Hoare logic for total correctness, except that we have added $/\rho$ at the end of each specification to indicate the ubiquitous procedure environment.

Skip:

$$\frac{}{[q]\ \mathbf{skip}\ [q]\ /\rho}$$

Abort:

$$\frac{}{[\mathbf{false}]\ \mathbf{abort}\ [q]\ /\rho}$$

Assignment:

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

Sequence:

$$\frac{[p]\ c_1\ [r]\ /\rho, \quad [r]\ c_2\ [q]\ /\rho}{[p]\ c_1\ ;\ c_2\ [q]\ /\rho}$$

Conditional:

$$\frac{\begin{array}{c}[r_1]\ c_1\ [q]\ /\rho \\ [r_2]\ c_2\ [q]\ /\rho\end{array}}{\begin{array}{c}[AB\ b => ab\_pre\ b\ r_1 \mid ab\_pre\ b\ r_2] \\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi}\ [q]\ /\rho\end{array}}$$

Iteration:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho \\ WF_c\ (\mathbf{assert}\ a\ \mathbf{with}\ v < x \\ \quad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ g\ \rho \\ [p]\ c\ [a \wedge (v < x)]\ /\rho \\ \{a \wedge (AB\ b) \wedge (v = x) \Rightarrow ab\_pre\ b\ p\} \\ \{a \wedge \sim(AB\ b) \Rightarrow ab\_pre\ b\ q\}\end{array}}{\begin{array}{c}[a]\ \mathbf{assert}\ a\ \mathbf{with}\ v < x \\ \quad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od}\ [q]\ /\rho\end{array}}$$

Rule of Adaptation:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho, \quad WF_c\ c\ g\ \rho, \quad WF_{xs}\ x, \quad DL\ x \\ x_0 = logicals\ x, \quad x_0' = variants\ x_0\ (FV_a\ q) \\ FV_c\ c\ \rho \subseteq x, \quad FV_a\ pre \subseteq x, \quad FV_a\ post \subseteq (x \cup x_0) \\ [x_0 = x \wedge pre]\ c\ [post]/\rho\end{array}}{[\ pre \wedge ((\forall x.\ (post \lhd [x_0'/x_0] \Rightarrow q)) \lhd [x/x_0'])\ ]\ c\ [q]/\rho}$$

Procedure Call:

$$\frac{\begin{array}{c}WF_{env}\ \rho, \quad WF_c\ (\mathbf{call}\ p(xs;es))\ g\ \rho \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (FV_a\ q \cup SL\ (xs\ \&\ glbs)), \quad y = vars\ \&\ vals\ \&\ glbs \\ u = xs\ \&\ vals', \quad v = vars\ \&\ vals, \quad x = xs\ \&\ vals'\ \&\ glbs \\ x_0 = logicals\ x, \quad y_0 = logicals\ y, \quad x_0' = variants\ x_0\ (FV_a\ q)\end{array}}{\begin{array}{c}[\ (pre \lhd [u/v] \wedge ((\forall x.\ (post \lhd [u, x_0'/v, y_0] \Rightarrow q)) \lhd [x/x_0'])) \lhd [vals' := es]\ ] \\ \mathbf{call}\ p(xs;es)[q]\ /\rho\end{array}}$$

Table 6.17: Hoare Logic for Total Correctness.

This must be used to resolve the semantics of procedure call. However, the environment ρ never changes during the execution of the program, and hence could be deleted from every specification, being understood in context. Of particular interest are the Rule of Adaptation and the Procedure Call Rule. Each rule has been proved completely sound from the corresponding rules in Tables 6.6 and 6.15, using the following rule:

$$\frac{\begin{array}{c}\{p\}\ c\ \{q\}\ /\rho \\ [p]\ c \Downarrow /\rho\end{array}}{[p]\ c\ [q]\ /\rho}$$

The procedure environment ρ is defined to be *well-formed for correctness* if for every procedure p, its body is totally correct with respect to the given precondition and postcondition:

$$\begin{array}{lll} WF_{env\_correct}\ \rho & = & \forall p.\ \ \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \mathbf{in} \\ & & \mathbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \mathbf{in} \\ & & \mathbf{let}\ x_0 = logicals\ x\ \mathbf{in} \\ & & [x_0 = x\ \wedge\ pre]\ c\ [post]\ /\rho \end{array}$$

An environment ρ is well-formed for correctness if and only if it is well-formed for partial correctness and for termination.

$$WF_{env\_correct}\ \rho\ =\ WF_{env\_partial}\ \rho\ \wedge\ WF_{env\_total}\ \rho$$

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.

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

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

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

$$\begin{array}{l}
vcg1\ (c_1\,;\,c_2)\ calls\ q\ \rho\ = \\
\qquad\qquad \mathbf{let}\ (s, h_2) = vcg1\ c_2\ calls\ q\ \rho\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (p, h_1) = vcg1\ c_1\ calls\ s\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad p,\ h_1 \,\&\, h_2
\end{array}$$

$$\begin{array}{l}
vcg1\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ calls\ q\ \rho\ = \\
\qquad\qquad \mathbf{let}\ (r_1, h_1) = vcg1\ c_1\ calls\ q\ \rho\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (r_2, h_2) = vcg1\ c_2\ calls\ q\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad (AB\ b \Rightarrow ab_pre\ b\ r_1 \mid ab_pre\ b\ r_2),\ h_1 \,\&\, h_2
\end{array}$$

$$\begin{array}{l}
vcg1\ (\mathbf{assert}\ a\ \mathbf{with}\ a_{pr}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ calls\ q\ \rho\ = \\
\qquad\qquad \mathbf{let}\ (v_0, v_1) = dest_{<}\ a_{pr}\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ (p, h) = vcg1\ c\ calls\ (a \wedge a_{pr})\ \rho\ \mathbf{in} \\
\qquad\qquad\qquad a,\ [\ a \wedge AB\ b \wedge (v_0 = v_1) \Rightarrow ab_pre\ b\ p\ ; \\
\qquad\qquad\qquad\quad a \wedge \sim\!(AB\ b) \Rightarrow ab_pre\ b\ q\ ] \,\&\, h
\end{array}$$

$$\begin{array}{l}
vcg1\ (\mathbf{call}\ p\ (xs\,;\,es))\ calls\ q\ \rho\ = \\
\qquad\qquad \mathbf{let}\ (vars, vals, glbs, pre, post, calls', rec, c) = \rho\ p\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ vals' = variants\ vals\ (FV_a\ q \cup SL(xs \,\&\, glbs))\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ u = xs \,\&\, vals'\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ v = vars \,\&\, vals\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ x = u \,\&\, glbs\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ y = v \,\&\, glbs\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ x_0 = logicals\ x\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ y_0 = logicals\ y\ \mathbf{in} \\
\qquad\qquad \mathbf{let}\ x_0' = variants\ x_0\ (FV_a\ q)\ \mathbf{in} \\
\qquad\qquad\qquad (\ (\ (pre \wedge calls\ p) \lhd [u/v]\ ) \wedge \\
\qquad\qquad\qquad\quad (\ (\forall x.\ (post \lhd [u \,\&\, x_0'/v \,\&\, y_0]) \Rightarrow q) \lhd [x/x_0']\ ) \\
\qquad\qquad\qquad) \lhd [vals' := es],\ [\,]
\end{array}$$

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

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

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` $\rightarrow$ `prog_env` $\rightarrow$ `aexp` $\rightarrow$ `env` $\rightarrow$ (`aexp` $\times$ (`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` $\rightarrow$ `cmd` $\rightarrow$ `prog_env` $\rightarrow$ `env` $\rightarrow$ (`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` $\rightarrow$ `env` $\rightarrow$ (`aexp`)`list`. *vcgd* takes a declaration and a procedure environment, and returns a list of verification conditions that must be proved in order to verify that

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

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

$$
\begin{array}{l}
extend\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ n\ p' \ = \\
\quad \textbf{let}\ q_1 = call\_progress\ p'\ p\ q\ \rho\ \textbf{in} \\
\quad (q_1 = \textbf{true}\ \ => \ [\,] \\
\quad\quad |\ p' = p_0\ \ => \\
\quad\quad\quad \textbf{let}\ (vars, vals, glbs, pre, post, calls, rec, c) = \rho\ p_0\ \textbf{in} \\
\quad\quad\quad\quad [\ pre \wedge induct\_pre\ rec \Rightarrow q_1\ ] \\
\quad\quad |\ p' \in SL(CONS\ p\ ps)\ \ => [\ pcs\ p' \Rightarrow q_1\ ] \\
\quad\quad |\ \ fan\_out\_graph\_vcs\ p'\ (CONS\ p\ ps)\ p_0\ q_1\ (pcs[q_1/p'])\ \rho\ all\_ps\ n \\
\quad) \\
\\
fan\_out\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ (n+1)\ \ = \\
\quad FLAT\ (MAP\ (extend\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ n)\ all\_ps) \\
\\
fan\_out\_graph\_vcs\ p\ ps\ p_0\ q\ pcs\ \rho\ all\_ps\ 0\ \ = \ [\,]
\end{array}
$$

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.

$$
\begin{array}{l}
graph\_vcs\ all\_ps\ \rho\ p\ = \\
\qquad \mathbf{let}\ (vars, vals, glbs, pre, post, calls, rec, c) = \rho\ p\ \mathbf{in} \\
\qquad fan\_out\_graph\_vcs\ p\ [\,]\ p\ rec\ (\lambda p'.\ \mathbf{true})\ \rho\ all\_ps\ (LENGTH\ all\_ps)
\end{array}
$$

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

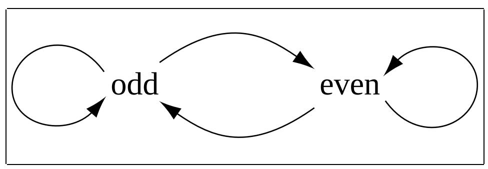

Figure 7.7: Procedure Call Graph for Odd/Even Example.

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.

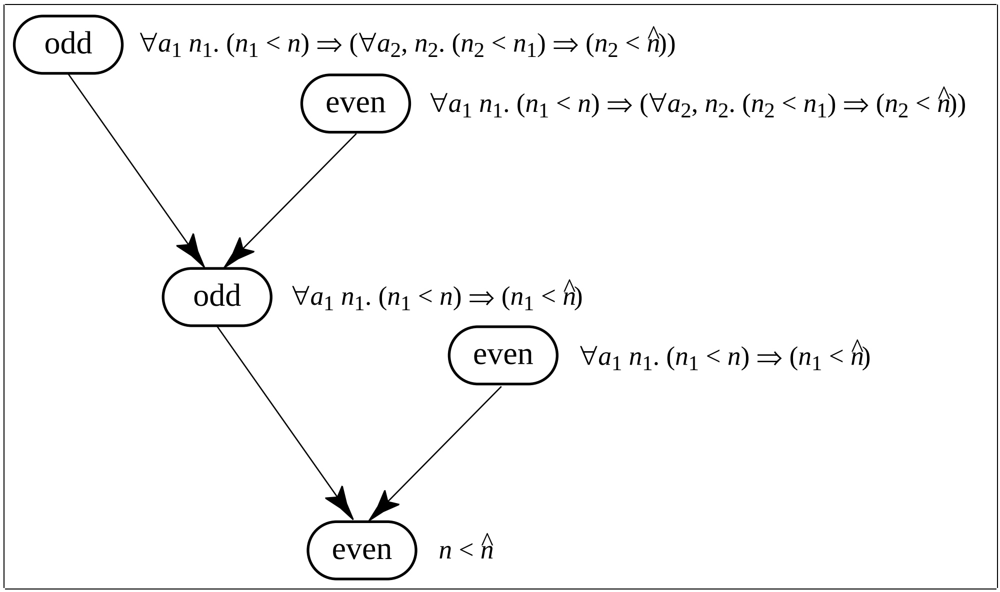

Figure 7.8: Procedure Call Tree for Odd/Even Example.

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

$$\begin{array}{rl} & \mathit{call\_progress}\ \mathit{odd}\ \mathit{odd}\ (\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n}))\ \rho \\ = & \forall a_1, n_1.\ (n_1 < n) \Rightarrow (\forall a_2, n_2.\ (n_2 < n_1) \Rightarrow (n_2 < \widehat{n})) \end{array}$$

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

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

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

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

which in this case is

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

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

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

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.

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

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.

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

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

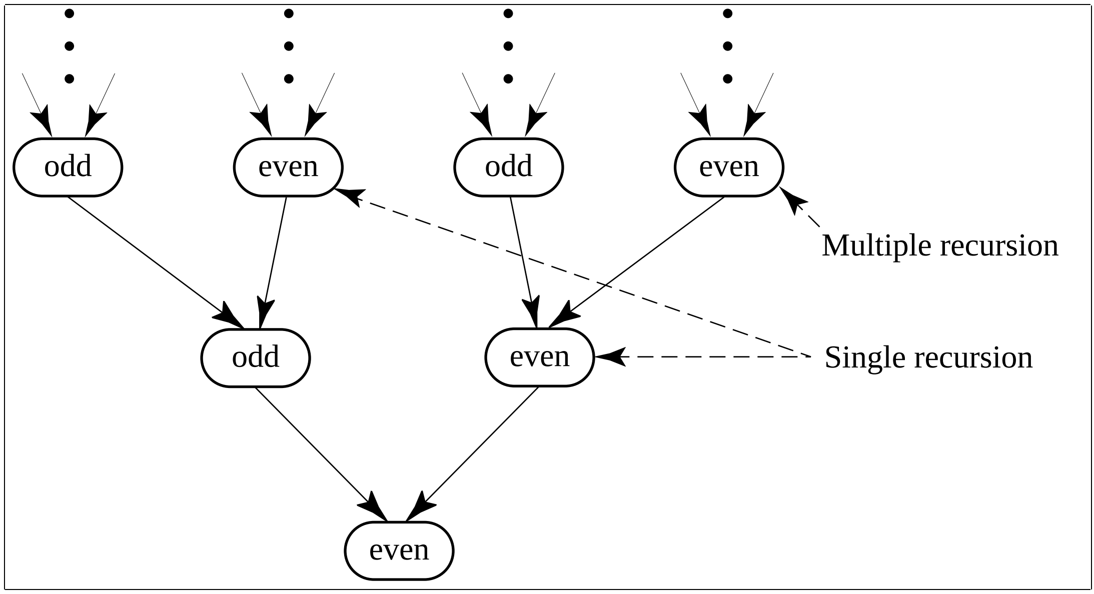

Figure 7.12: Procedure Call Tree for Recursion for Odd/Even Example.

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*.

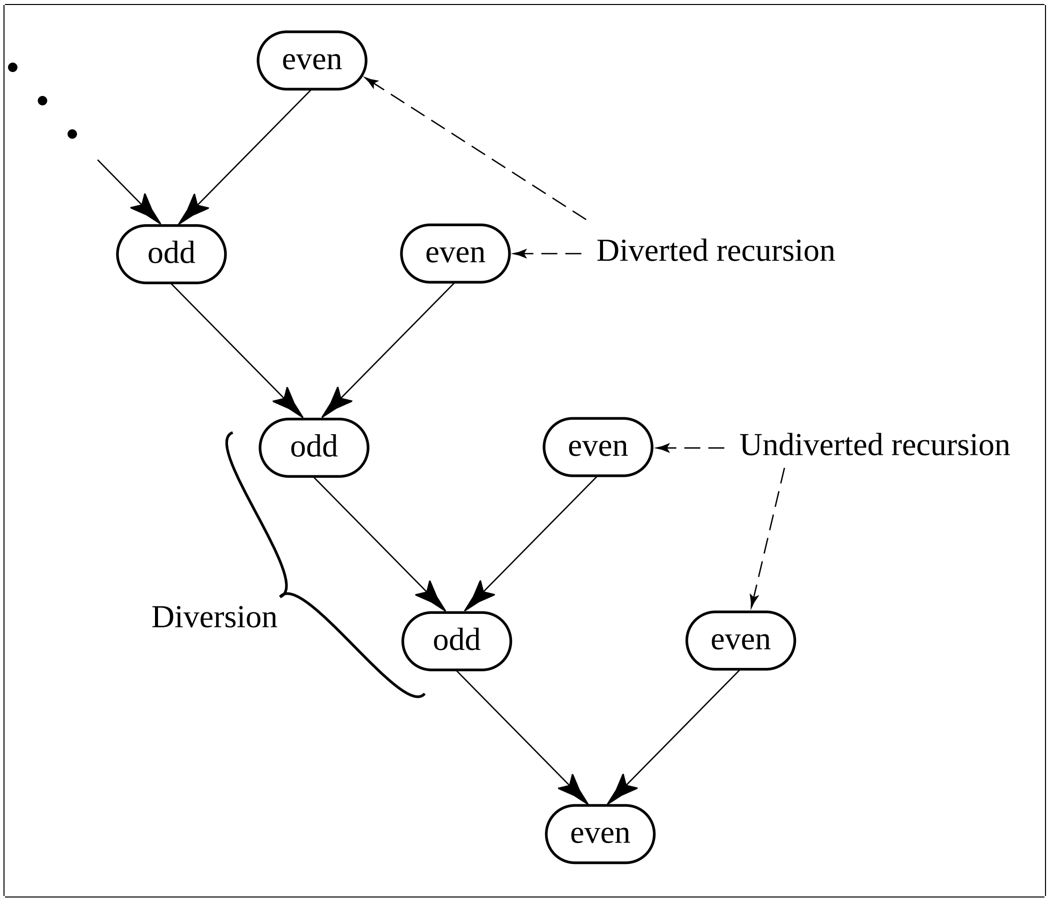

Figure 7.13: Procedure Call Tree for Single Recursion for Odd/Even Example.

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

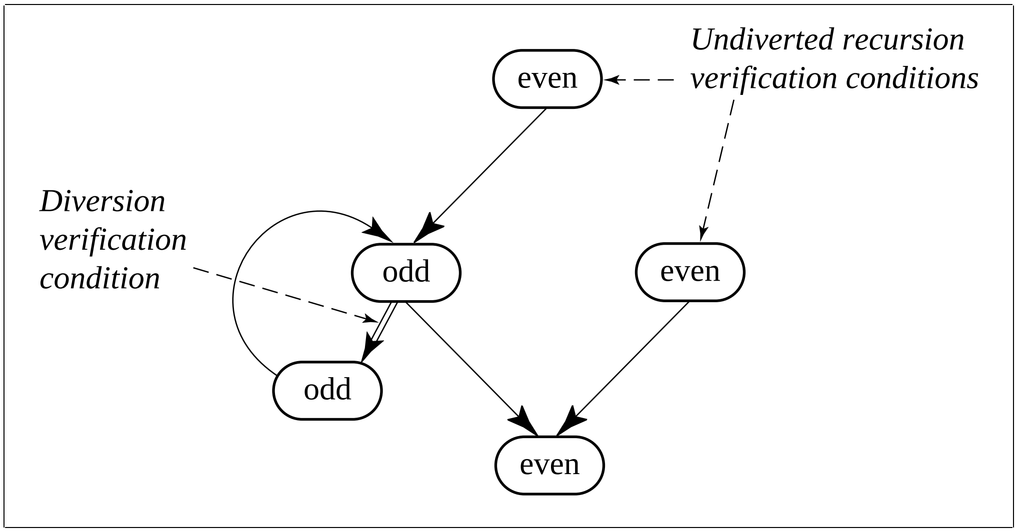

Figure 7.14: Diverted and Undiverted Verification Conditions for Odd/Even.

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 $vcg1$ function are listed in Table 7.1. There are seven theorems listed, which correspond to seven ways that the results of the $vcg1$ 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 $vcg1$ 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`	$\forall c\ calls\ q\ \rho.\ \ WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow \{p\}\ c\ \{q\}\ /\rho,\ 0)$
`vcg1_k_THM`	$\forall c\ calls\ q\ \rho\ k.\ WF_{envk}\ \rho\ k\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow \{p\}\, c\, \{q\}/\rho, k+1)$
`vcg1p_THM`	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow \{p\}\ c\ \{q\}\ /\rho)$
`vcg1_PRE_PROGRESS`	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow \{p\}\ c \longrightarrow$ **pre** $/\rho)$
`vcg1_BODY_PROGRESS`	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_{calls}\ calls\ \rho$ $\quad \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow \{p\}\ c \longrightarrow calls\ /\rho)$
`vcg1_TERM`	$\forall c\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**all_el close** $h \Rightarrow [p]\ c \downarrow /\rho)$
`vcg1_THM`	$\forall c\ calls\ q\ \rho.\ \ WF_{env}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **let** $(p,h) = vcg1\ c\ calls\ q\ \rho$ **in** $\quad$ (**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`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad \{p\}\ c\ \{q\}\ /\rho,\ 0$
`vcgc_k_THM`	$\forall c\ p\ calls\ q\ \rho\ k.\ WF_{envk}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad \{p\}\ c\ \{q\}\ /\rho,\ k+1$
`vcgcp_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad \{p\}\ c\ \{q\}\ /\rho$
`vcgc_PRE_PROGRESS`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad \{p\}\ c \rightarrow \textbf{pre}\ /\rho$
`vcgc_BODY_PROGRESS`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad \{p\}\ c \rightarrow calls\ /\rho$
`vcgc_TERM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\quad [p]\ c \downarrow /\rho$
`vcgc_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{env}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ $\quad$ **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 $fan\_out\_graph\_vcs$.

$$
\begin{array}{l}
\forall p\ \rho\ all\_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c.\\
\quad WF_{env\_syntax}\ \rho\ \wedge\\
\quad WF_{env\_pre}\ \rho\ \wedge\\
\quad WF_{env\_calls}\ \rho\ \wedge\\
\quad (\forall p'.\ (p' \notin SL\ all\_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge\\
\quad p \in SL\ all\_ps\ \wedge\\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge\\
\quad \textbf{all\_el close}\ (graph\_vcs\ all\_ps\ \rho\ p)\\
\quad \Rightarrow\\
\\
\quad (\forall ps.\\
\qquad\quad DL\ (ps\,\&\,[p])\ \Rightarrow\\
\qquad\quad \textbf{close}\ (pre \wedge induct\_pre\ rec \Rightarrow\\
\qquad\qquad\qquad call\_path\_progress\ p\ ps\ p\ rec\ \rho))\ \wedge\\
\quad (\forall p'\ ps\ ps_1\ ps_2.\\
\qquad\quad (p' \neq p)\ \wedge\\
\qquad\quad DL\ ps\ \wedge\\
\qquad\quad p \notin\ SL\ ps\ \wedge\\
\qquad\quad ps = ps_2\,\&\,(CONS\ p'\ ps_1)\ \Rightarrow\\
\qquad\quad \textbf{close}\ (call\_path\_progress\ p'\ ps_1\ p\ rec\ \rho \Rightarrow\\
\qquad\qquad\qquad call\_path\_progress\ p'\ (ps_2\,\&\,(CONS\ p'\ ps_1))\ p\ rec\ \rho))
\end{array}
$$

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-

$$
\begin{array}{l}
\forall ps\ p\ \rho\ all\_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c.\\
\quad WF_{env\_syntax}\ \rho\ \wedge\\
\quad WF_{env\_pre}\ \rho\ \wedge\\
\quad WF_{env\_calls}\ \rho\ \wedge\\
\quad (\forall p'.\ (p' \notin SL\ all\_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge\\
\quad p \in SL\ all\_ps\ \wedge\\
\quad p \notin SL\ ps\ \wedge\\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge\\
\quad \textbf{all\_el close}\ (graph\_vcs\ all\_ps\ \rho\ p)\\
\quad \Rightarrow\\
\quad \{pre \wedge induct\_pre\ rec\}\ p \mathrel{—} ps \rightarrow p\ \{rec\}\ /\rho
\end{array}
$$

Table 7.7: Theorem of verification of all single recursion.

$$
\begin{array}{l}
\forall n\ ps\ p\ \rho\ all\_ps\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c.\\
\quad WF_{env\_syntax}\ \rho\ \wedge\\
\quad WF_{env\_pre}\ \rho\ \wedge\\
\quad WF_{env\_calls}\ \rho\ \wedge\\
\quad (\forall p'.\ (p' \notin SL\ all\_ps) \Rightarrow (\rho\ p' = \rho_0\ p'))\ \wedge\\
\quad LENGTH\ ps = n\ \wedge\\
\quad p \in SL\ all\_ps\ \wedge\\
\quad \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \wedge\\
\quad \textbf{all\_el close}\ (graph\_vcs\ all\_ps\ \rho\ p)\\
\quad \Rightarrow\\
\quad \{pre \wedge induct\_pre\ rec\}\ p \mathrel{—} ps \rightarrow p\ \{rec\}\ /\rho
\end{array}
$$

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.

$$\forall \pi\ q.\ \ WF_p\ \pi\ \wedge\ \textbf{all\_el close}\ (vcg\ \pi\ q)\ \Rightarrow\ \pi\ [q]$$

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.

CHAPTER 8

Example Runs

"By their fruits you shall know them."

— Matthew 7:20

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

— Hebrews 6:12

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

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

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

8.1 Quotient/Remainder

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

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

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

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

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

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

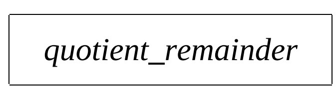

Figure 8.1: Procedure Call Graph for Quotient/Remainder Program.

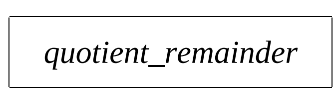

Figure 8.2: Procedure Call Tree for root procedure *quotient_remainder*.

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

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

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

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

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

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

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

By the "ASSIGN" rule, we have

```
[[ {^x = 0 * ^y + r /\ 0 < y /\ ^y = y}
     q := 0
   {^x = q * ^y + r /\ 0 < y /\ ^y = y} ]]
```

By the "ASSIGN" rule, we have

```
[[ {^x = 0 * ^y + x /\ 0 < y /\ ^y = y}
     r := x
   {^x = 0 * ^y + r /\ 0 < y /\ ^y = y} ]]
```

By the "SEQ" rule, we have

```
[[ {^x = 0 * ^y + x /\ 0 < y /\ ^y = y}
     r := x; q := 0
   {^x = q * ^y + r /\ 0 < y /\ ^y = y} ]]
```

By the "SEQ" rule, we have

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

with verification conditions

```
"[[[ {((^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(r < y)) /\
      r = ^r ==>
      (^x = (q + 1) * ^y + (r - y) /\ 0 < y /\ ^y = y) /\
      r - y < ^r} ]];
  [[ {(^x = q * ^y + r /\ 0 < y /\ ^y = y) /\ ~(~(r < y)) ==>
      ^x = q * ^y + r /\ r < ^y} ]]]"
```

By precondition strengthening, we have

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

with additional verification condition

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

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure quotient_remainder:

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

For the main body,

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

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

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

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

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

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

() : void
```

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

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

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

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

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

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

8.2 McCarthy's "91" Function

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

We define the function $f91$ as

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

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

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

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

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

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

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

        p91(a; 77)

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

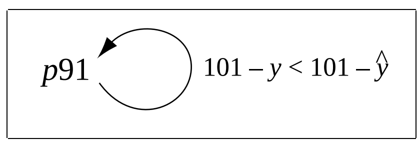

Figure 8.3: Procedure Call Graph for McCarthy's "91" Program.

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

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

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

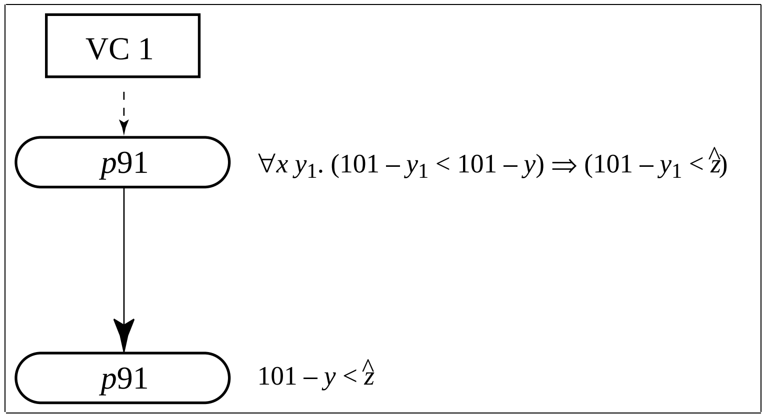

Figure 8.4: Procedure Call Tree for root procedure $p91$.

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

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

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

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

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

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

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure p91:

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

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

For the main body,

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

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

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

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

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

() : void
```

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

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

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

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

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

8.3 Odd/Even Mutual Recursion

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

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

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

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

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

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

        odd(a; 5)

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

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

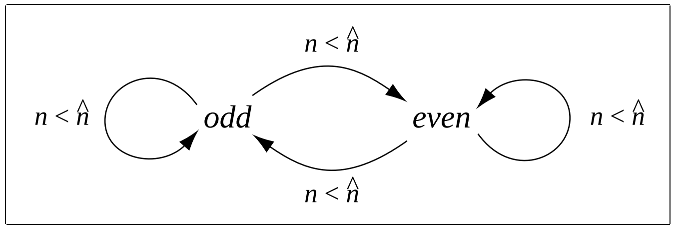

Figure 8.5: Procedure Call Graph for Odd/Even Program.

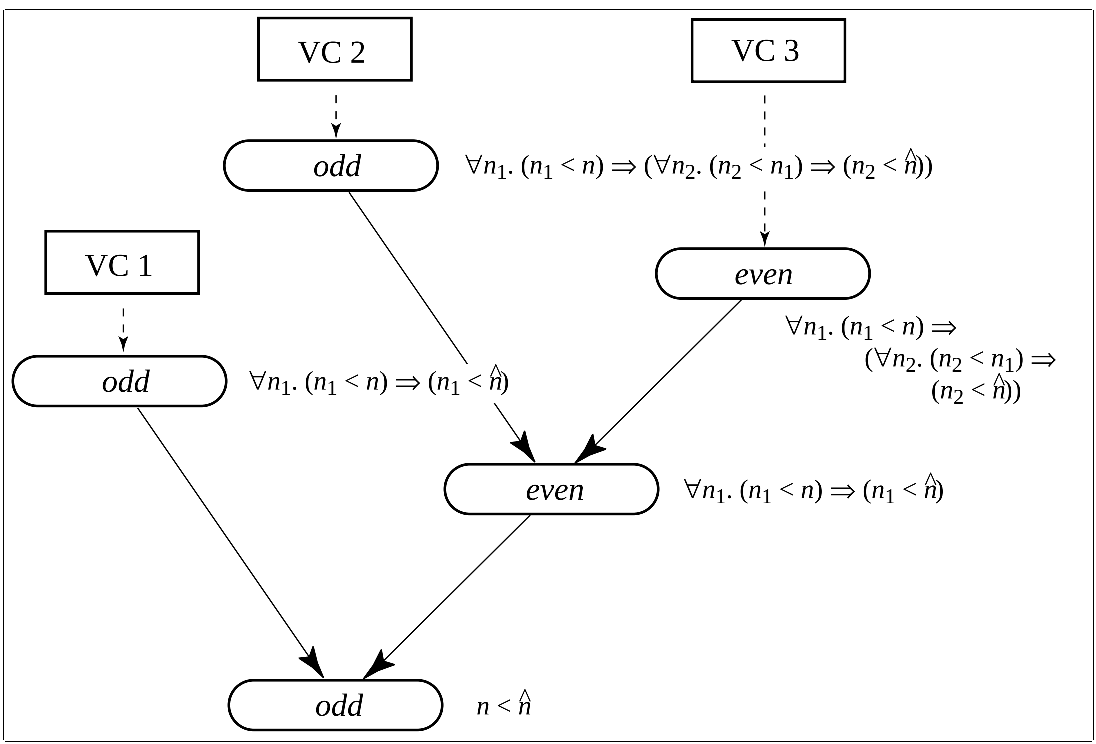

Figure 8.6: Procedure Call Tree for root procedure *odd*.

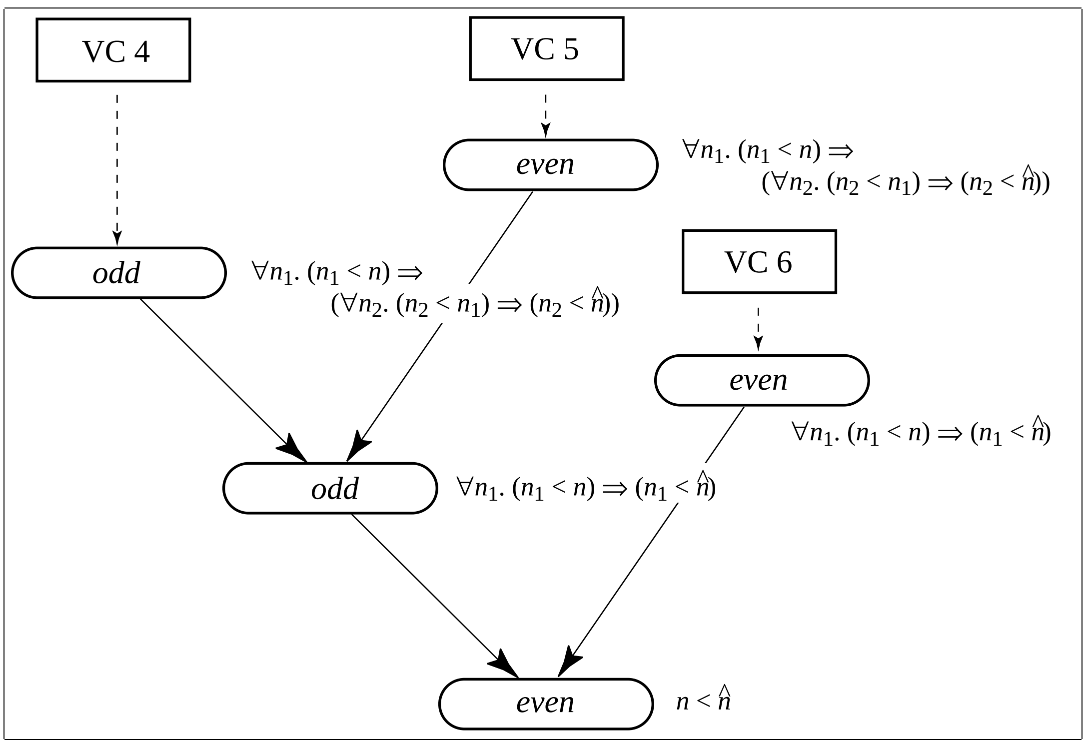

Figure 8.7: Procedure Call Tree for root procedure *even*.

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

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

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

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

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

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

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

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

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

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

For procedure `even`,

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

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

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

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

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

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

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

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure even:

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

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

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

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

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

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

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

Traversing the call graph back from the procedure odd:

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

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

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

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

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

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

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

For the main body,

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

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

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

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

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

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

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

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

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

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

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

() : void
```

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

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

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

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

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

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

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

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

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

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

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

8.4 Pandya and Joseph's Product Procedures

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

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

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

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

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

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

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

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

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

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

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

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

Figure 8.8: Pandya and Joseph's Product Procedures.

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

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

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

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

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

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

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

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

    if $y = 0$
    then $\boxed{z + x * y = a * b \; \wedge \; y = 0}$
        skip
        $\boxed{z = a * b}$
    else $\boxed{z + x * y = a * b \; \wedge \; 0 < y \leq i \; \wedge \; even(y)}$
        $\boxed{z + x * y = a * b \; \wedge \; (y \textbf{ div } 2) \leq i - 1}$
        $x := 2 * x$;
        $y := y$ **div** 2;
        $\boxed{z + x * y = a * b \; \wedge \; y \leq i - 1} \ldots \boxed{q_p(i - 1)}$
        $product(;)$
        $\boxed{z = a * b}$
    fi
end procedure;

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

To motivate this proof, Pandya and Joseph state

> On each successive call to the procedure *product* the value of y becomes y **div** 2. Thus we can argue that the procedure *product* terminates because on each successive call to *product*, the value of y decreases, and if a call to *product* is made with $y = 0$ then no further recursive call to *product* is made. It is possible to give a simple total correctness proof based on the above argument using induction over the number of calls to *product* active at any instant.

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

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

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

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

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

$evenproduct(;)$

$\boxed{z = a * b}$

else $\boxed{z + x * y = a * b \ \wedge \ odd(y) \ \wedge \ y = \hat{y}}$

$oddproduct(;)$

$\boxed{z = a * b}$

fi

end procedure;

```
procedure oddproduct(; );
      global    x, y, z, a, b;
      pre       z + x * y = a * b ∧ odd(y);
      post      z = a * b;
      calls     evenproduct    with y < ŷ;
      recurses                 with y < ŷ;
```

$\boxed{(z + x) + x * (y - 1) = a * b \ \wedge \ even(y - 1) \ \wedge \ y - 1 < \hat{y}}$

$y := y - 1;$

$z := z + x;$

$\boxed{z + x * y = a * b \ \wedge \ even(y) \ \wedge \ y < \hat{y}}$

$evenproduct(;)$

$\boxed{z = a * b}$

end procedure;

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

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

$$\boxed{\begin{array}{lll} y = 0 & => & z = a * b \\ & \mid & z + (2 * x) * (y \ \mathbf{div}\ 2) = a * b \ \wedge \ y \ \mathbf{div}\ 2 < \widehat{y} \end{array}}$$

  if $y = 0$
  then $\boxed{z = a * b}$
    skip
    $\boxed{z = a * b}$
  else $\boxed{z + (2 * x) * (y \ \mathbf{div}\ 2) = a * b \ \wedge \ y \ \mathbf{div}\ 2 < \widehat{y}}$
    $x := 2 * x$;
    $y := y \ \mathbf{div}\ 2$;
    $\boxed{z + x * y = a * b \ \wedge \ y < \widehat{y}}$
    product(;)
    $\boxed{z = a * b}$
  fi
end procedure;

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

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

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

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

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

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

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

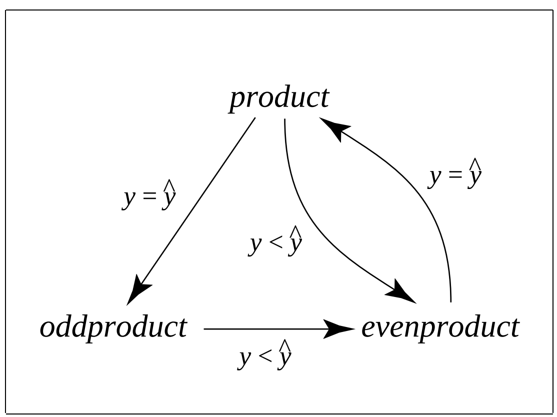

Figure 8.13: Procedure Call Graph for Pandya and Joseph's Product Program.

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

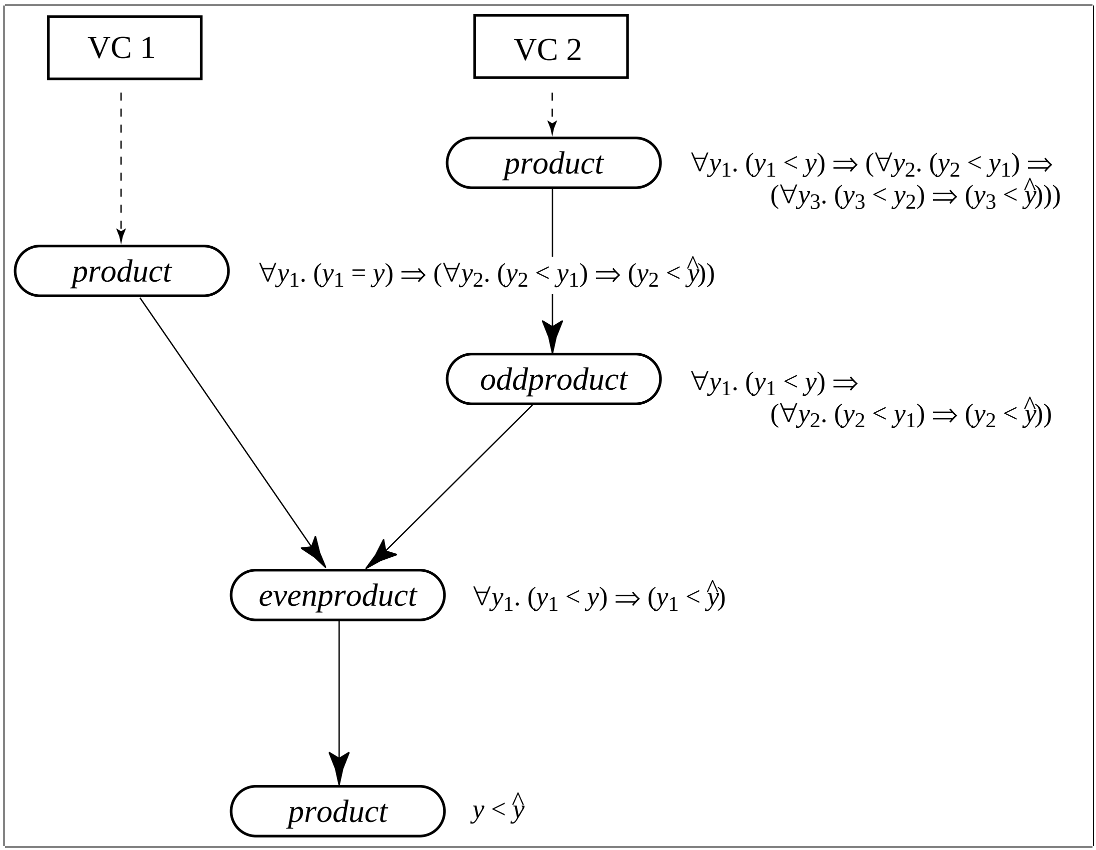

Figure 8.14: Procedure Call Tree for root procedure *product*.

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

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

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

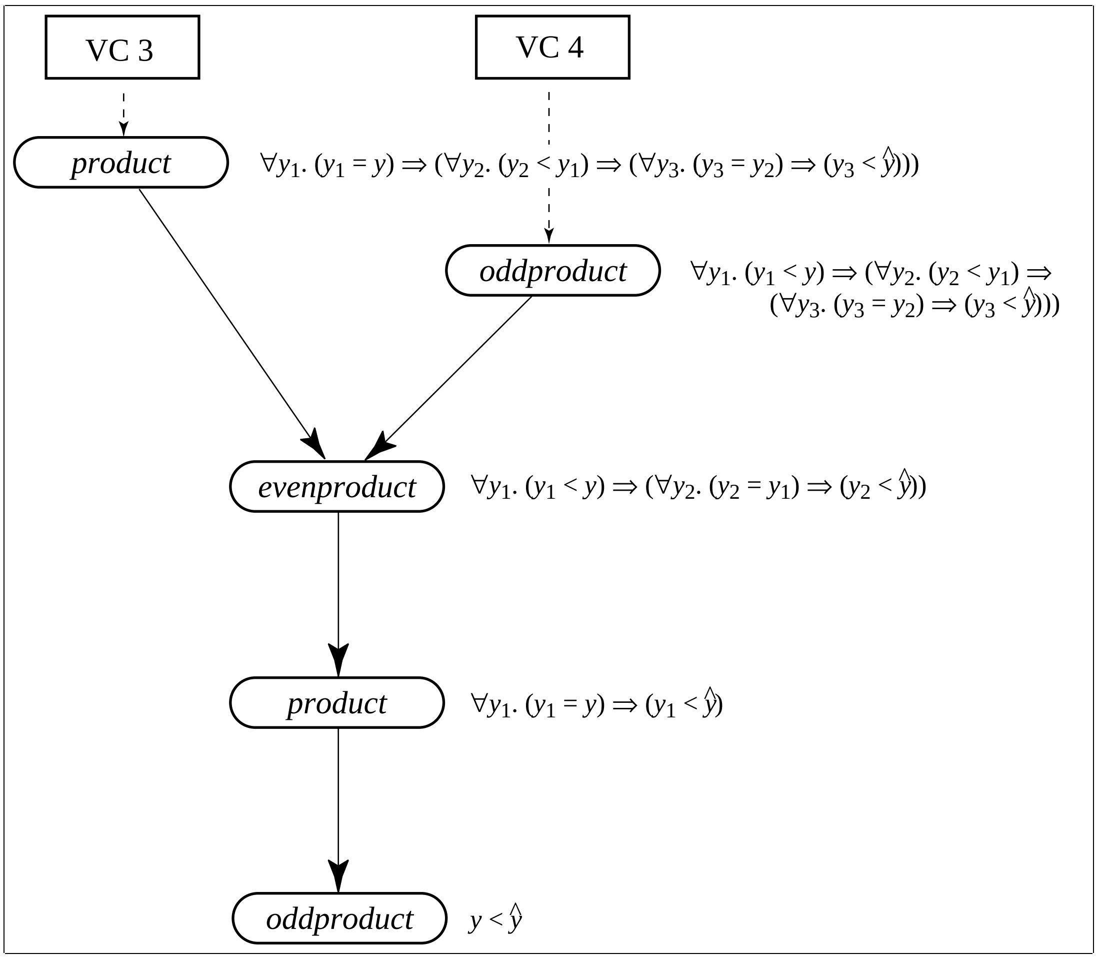

Figure 8.15: Procedure Call Tree for root procedure *oddproduct*.

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

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

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

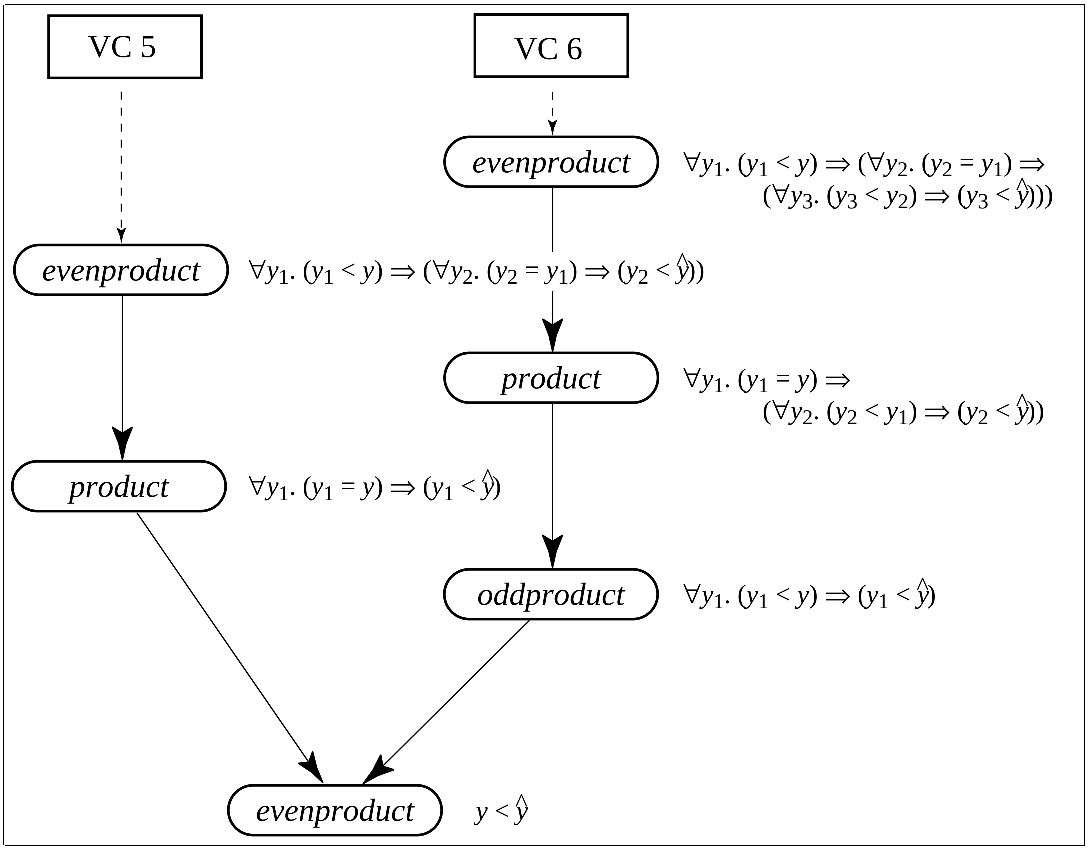

Figure 8.16: Procedure Call Tree for root procedure *evenproduct*.

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

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

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

8.5 Cycling Termination

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

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

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

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

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

```
g [[ program

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

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

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

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

        pedal(;7,12)

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

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

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

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

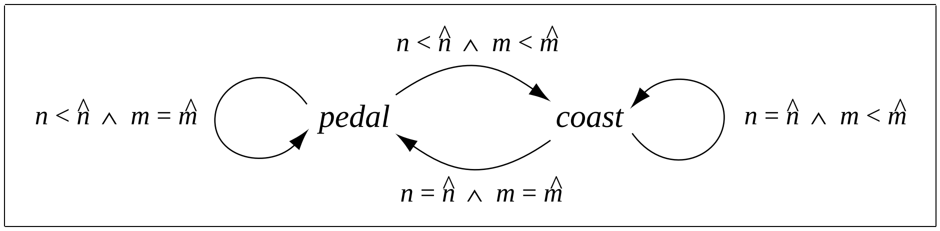

Figure 8.17: Procedure Call Graph for Cycling Termination Program.

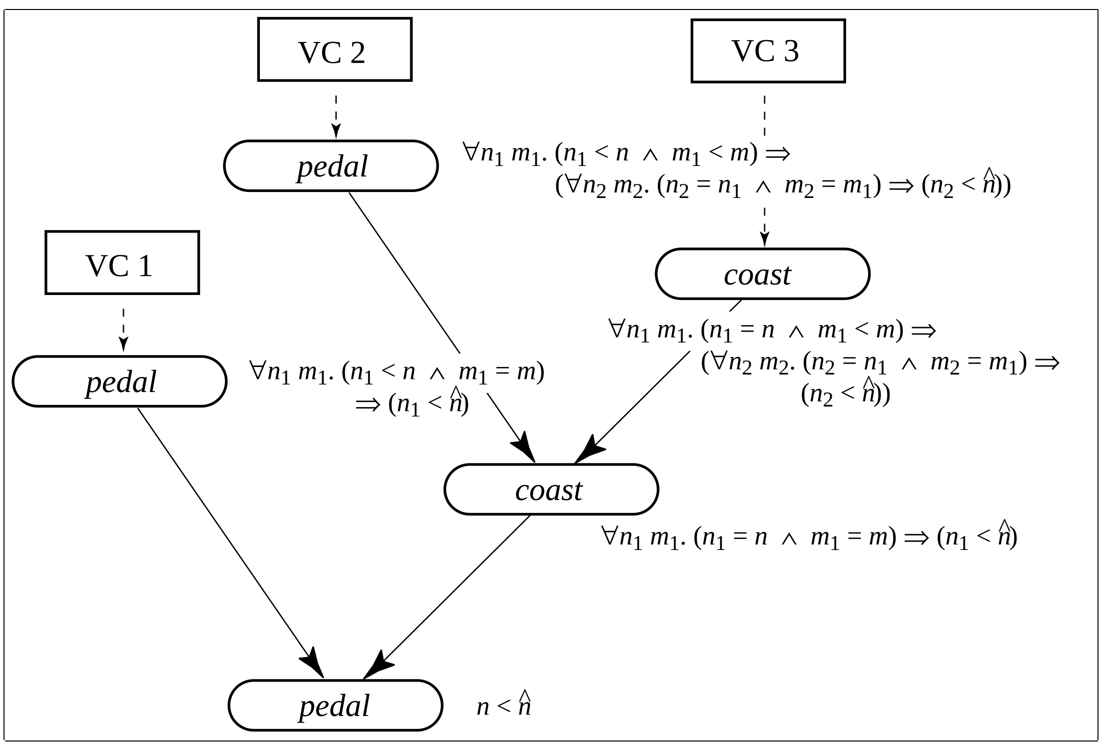

Figure 8.18: Procedure Call Tree for root procedure *pedal*.

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

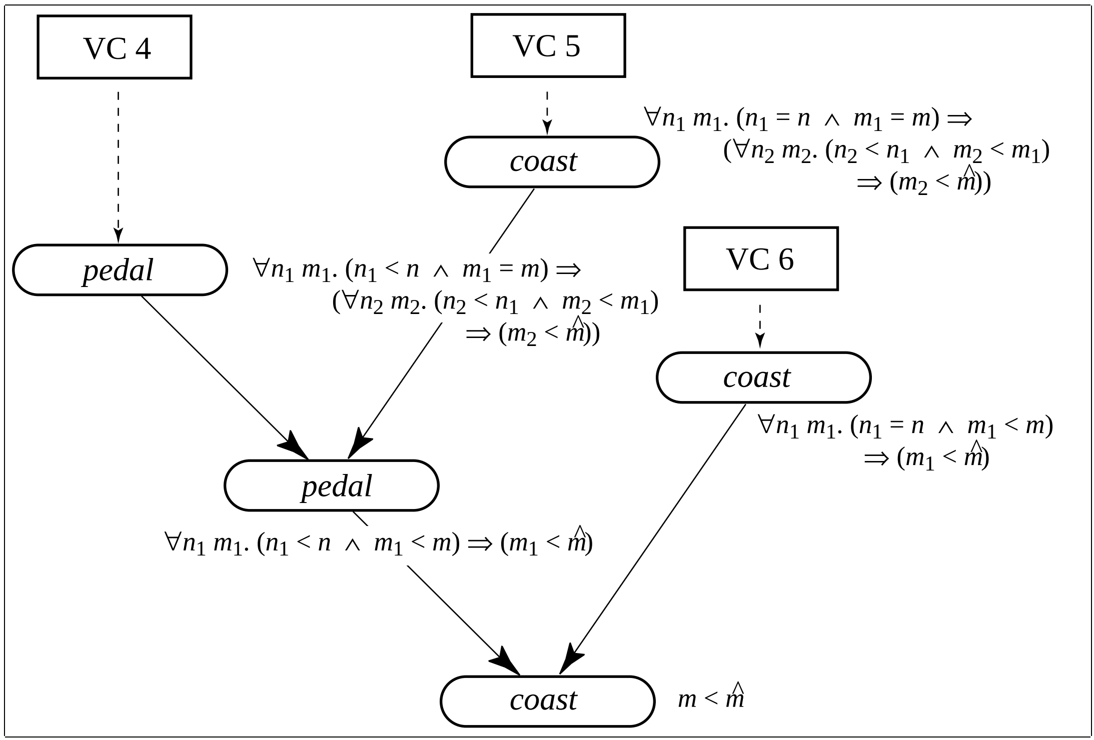

Figure 8.19: Procedure Call Tree for root procedure *coast*.

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

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

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

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

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

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

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

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

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

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

For procedure `coast`,

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

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

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

By the "CALL" rule, we have

```
   [[ {(true /\ n = ^n /\ m = ^m) /\
       (!n1 m1. true ==> (0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                                   (!n m. true ==> true) | true))}
       pedal(;n,m)
      {(0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                 (!n m. true ==> true) | true)} ]]
```

By the "SEQ" rule, we have

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

By precondition strengthening, we have

```
   [[ {(^n = n /\ ^m = m /\ true) /\ true}
       pedal(;n,m); if 0 < m then coast(;n,m - 1) else skip fi
      {true} ]]
```

with additional verification condition

```
   [[ {(^n = n /\ ^m = m /\ true) /\ true ==>
       (true /\ n = ^n /\ m = ^m) /\
       (!n1 m1. true ==> (0 < m => (true /\ n = ^n /\ m - 1 < ^m) /\
                                   (!n m. true ==> true) | true))} ]]
```

Examining the structure of the procedure call graph:

Traversing the call graph back from the procedure coast:

By the call graph progress from procedure coast to coast, we have

```
   [[ {true /\ (!n1 m1. n1 = n /\ m1 < m ==> m1 < ^m)}
      coast-<>->coast
      {m < ^m} ]]
```

Generating the undiverted recursion verification condition

```
   [[ {true /\ m = ^m ==> (!n1 m1. n1 = n /\ m1 < m ==> m1 < ^m)} ]]
```

By the call graph progress from procedure pedal to coast, we have

```
   [[ {true /\ (!n1 m1. n1 < n /\ m1 < m ==> m1 < ^m)}
      pedal-<>->coast
      {m < ^m} ]]
```

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

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

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

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

Traversing the call graph back from the procedure pedal:

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

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

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

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

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

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

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

For the main body,

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

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

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

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

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

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

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

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

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

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

() : void
```

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

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

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

- The diversion of *pedal* in $pedal \rightarrow pedal \rightarrow coast$ does not interfere with the recursive progress of the procedure *coast* (VC4).

- The value of the recursion expression of the procedure *coast* strictly decreases across the undiverted recursion path $coast \rightarrow pedal \rightarrow coast$ (VC5).

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

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

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

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

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

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

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

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

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

CHAPTER 9

Source Code

"A garden enclosed

Is my sister, my spouse,

A spring shut up,

A fountain sealed. ...

A fountain of gardens,

A well of living waters,

And streams from Lebanon."

— Song of Solomon 4:12, 15

The source code for the Sunrise system may be retrieved by anonymous ftp from Internet site `ftp.cs.ucla.edu`, in directory `/pub/homeier/sunrise`. It is based on version 2.02 of Higher Order Logic (HOL). It contains a modified version of one library, `ind_defs`, revised to work in HOL version 2.02.

There are altogether twenty-two theories. Their sizes are given in Table 9.1. The column headings have the following meanings:

Column:	Meaning
Theory Name:	the name of the HOL theory
Typs:	the number of new types declared in the theory
Defs:	the number of new constants declared in the theory
Thms:	the number of theorems proven and stored in the theory
ALL:	the sum of the preceeding three columns
TLen:	the length of the listing of the theory, in lines
ATL:	the average length of a theorem (def., etc.) as listed, in lines
SLen:	the length of the source file of the theory, in lines
ASL:	the avg. length of source for a theorem (def., etc.), in lines

Theory Name	Typs	Defs	Thms	ALL	TLen	ATL	SLen	ASL
more_finite_sets	0	1	37	38	160	4.2	1003	26.4
bindings	0	5	88	93	432	4.6	3107	33.4
variables	1	5	7	13	66	5.1	106	8.2
variants	0	13	54	67	273	4.1	1536	22.9
assert_syntax	2	23	10	35	601	17.2	85	2.4
assert_semantics	0	7	14	21	149	7.1	710	33.8
substitutions	0	7	56	63	355	5.6	2096	33.3
var_substitutions	0	4	93	97	573	5.9	4020	41.4
prog_syntax	5	33	25	63	622	9.9	105	1.7
prog_substitutions	0	7	25	32	208	6.5	651	20.3
prog_semantics	0	9	13	22	457	20.8	409	18.6
free_variables	0	6	36	42	249	5.9	2916	69.4
translations	0	16	21	37	213	5.8	805	21.8
progress	0	15	15	30	427	14.2	845	28.2
well_formed	0	34	79	113	916	8.1	4139	36.6
cmd_semantics	0	0	21	21	290	13.8	3382	161.0
hoare_rules	0	1	100	101	1030	10.2	8841	87.5
progress_rules	0	2	46	48	362	7.5	2925	60.9
semantic_stages	0	6	26	32	400	12.5	3803	118.8
stage_semantics	0	0	32	32	481	15.0	5427	169.6
termination	0	12	53	65	579	8.9	3085	47.5
vcg	0	11	55	66	702	10.6	7516	113.9
TOTALS	8	217	906	1131	9545	8.4	57512	50.9

Table 9.1: Sunrise Theory Sizes.

Part III

Tour of Interesting Aspects

CHAPTER 10

Partial Correctness

"Finally, brethren, whatever things are true, whatever things are notable, whatever things are just, whatever things are pure, whatever things are lovely, whatever things are of good report; if there is any virtue and if there is anything praiseworthy—meditate on these things."
— Philippians 4:8

In this chapter we present various interesting aspects of the VCG system, which support the proof of partial correctness of commands and the environment.

10.1 Variants

A variable is represented by a new concrete type `var`, with one constructor function, VAR: `string` $\rightarrow$ `num` $\rightarrow$ `var`. We define two deconstructor functions:

$$\begin{array}{rcl} Base(VAR\ str\ n) & = & str \\ Index(VAR\ str\ n) & = & n \end{array}$$

The number attribute eases the creation of variants of a variable, which are made by (possibly) increasing the number.

All possible variables are considered predeclared of type `num`. In future versions, we hope to treat other data types, by introducing a more complex state and a static semantics for the language which performs type-checking. We distinguish between program variables and logical variables; the latter cannot be changed by program control. In the Sunrise language, we denote logical variables by beginning its name with a caret character ('^'), as part of its string. A "well-formed" variable, such as used in normal program code, will not have this prefix.

The *variant* function has type `var` $\rightarrow$ (`var`)`set` $\rightarrow$ `var`. $variant\ x\ s$ returns a variable which is a variant of x, which is guaranteed not to be in the "exclusion" set s. If x is not in the set s, then it is its own variant. *variant* is used in defining proper substitution on quantified expressions.

The definition of *variant* is somewhat deeper than might originally appear. To have a constructive function for making variants in particular instances, we wanted

$$variant\ x\ s = (x \in s => variant\ (mk\_variant\ x\ 1)\ s \mid x), \tag{10.1}$$

where

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

For any finite set s, this definition of *variant* will terminate, but unfortunately, it is not primitive recursive on the set s, and so does not conform to the requirements of HOLs recursive function definition. As a substitute, we wanted to define the *variant* function by specifying its properties, as

$$(variant\ x\ s)\ is\_variant\ x, \text{ and} \tag{10.2}$$

$$variant\ x\ s \notin s, \text{ and} \tag{10.3}$$

$$\begin{aligned} &\forall z.\ z\ is\_variant\ x \wedge z \notin s\ \Rightarrow \\ &\quad Index(variant\ x\ s) \leq Index(z), \end{aligned} \tag{10.4}$$

where *is_variant* is an infix binary predicate, defined as

$$y\ is\_variant\ x = (Base(y) = Base(x)\ \wedge\ Index(x) \leq Index(y)).$$

But even the above specification did not easily support the proof of the existence theorem, that such a variant existed for any x and s, because the set of values for z satisfying the antecedent of property 10.4 is infinite, and we were working strictly with finite sets. The solution was to introduce the function *variant_set* of type `var` $\rightarrow$ `num` $\rightarrow$ (`var`)`set`, where $variant\_set\ x\ n$ returns the set of the first n variants of x, all different from each other. Then the cardinality of the set is n, i.e.,

$$CARD\ (variant\_set\ x\ n) = n.$$

The definition of *variant_set* is

$$\begin{aligned} variant\_set\ x\ 0 &= EMPTY \\ variant\_set\ x\ (n+1) &= (mk\_variant\ x\ n)\ INSERT\ (variant\_set\ x\ n), \end{aligned}$$

where $EMPTY$ is the empty set and $INSERT$ is the infix binary operator to add an element to a set, predefined in `HOL`.

Then by the pigeonhole principle, we are guaranteed tha there must be at least one variable in $variant\_set\ x\ (CARD\ s + 1)$ which is not in the set s. This led to the needed existence theorem. We then defined *variant* with the following properties:

$$(variant\ x\ s) \in variant\_set\ x\ (CARD\ s + 1),\ \text{and} \tag{10.5}$$

$$variant\ x\ s \notin s,\ \text{and} \tag{10.6}$$

$$\begin{array}{l}\forall z.\ z \in variant\_set\ x\ (CARD\ s + 1) \wedge z \notin s\ \Rightarrow \\ \quad Index(variant\ x\ s) \leq Index(z).\end{array} \tag{10.7}$$

From this definition, we then proved both the original set of properties (10.2–10.4), and also the constructive function definition 10.1, as theorems.

Finally, given the definition of *variant*, we defined a similar operator on lists:

$$\begin{array}{rcl} variants\ [\,]\ s & = & [\,] \\ variants\ (CONS\ x\ xs)\ s & = & \textbf{let}\ x' = variant\ x\ s\ \textbf{in} \\ & & CONS\ x'\ (variants\ xs\ (x'\ INSERT\ s)). \end{array}$$

This definition has the property that the resulting list has no duplicates. We say it is a "distinct list", according to the predicate DL, which is defined as follows.

$$\begin{array}{rcl} DL\ [\,] & = & \mathrm{T} \\ DL\ (CONS\ x\ xs) & = & x \notin (SL\ xs)\ \wedge\ DL\ xs \end{array}$$

Here SL is simply an operator to convert a list into a set, defined as follows.

$$\begin{array}{rcl} SL\ [\,] & = & EMPTY \\ SL\ (CONS\ x\ xs) & = & x\ INSERT\ (SL\ xs) \end{array}$$

10.2 Substitution

The concept of substitution at first appears very simple, but it actually can be a mine field of subtlety and misdirection. This subtlety arises primarily from the need to avoid the capture of free variables by bindings imposed by quantifiers in the expression receiving the substitution. Typically this is accomplished by the systematic renaming of the bound variables to preclude capturing the free variables of the expression being inserted. We have found an error in one published proof of the Substitution Lemma, and other researchers have shared their experience with the surprising difficulty of this area. The most thorough treatment we have found is by de Bakker in [dB80].

10.2.1 Assertion Language Expression Substitution

We define proper substitution on assertion language expressions using the technique of *simultaneous substitutions*, following Stoughton [Sto88]. The usual definition of proper substitution is a fully recursive function. Unfortunately, `HOL` only supports primitive recursive definitions. To overcome this, we use simultaneous substitutions, which are represented by functions of type `subst` = `var` $\rightarrow$ `aexp`. This describes a family of substitutions, all of which are considered to take place simultaneously. This family is in principle infinite, but in practice all but a finite number of the substitutions are the identity substitution ι. The virtue of this approach is that the application of a simultaneous substitution to an assertion language expression may be defined using only primitive recursion, not full recursion, and then the normal single substitution operation of $[v/x]$ may be defined as a special case:

$$[v/x] = \lambda y.\ (y = x => v \mid AVAR\ y).$$

We apply a substitution by the infix operator $\triangleleft$. Thus, $a \triangleleft ss$ denotes the application of the simultaneous substitution ss to the expression a. Therefore

$a \lhd [v/x]$ denotes the single substitution of the expression v for the variable x wherever x appears free in a.

While defining substitution on several different kinds of language phrases, we will add a subscript indicating the kind of phrase on which the substitution is being performed. For example, we will define $\lhd_v$ for substitutions on `vexp`, $\lhd_{vs}$ for substitutions on `(vexp)list`, and $\lhd_a$ for substitutions on `aexp`. However, outside this chapter we will usually simply use an undecorated $\lhd$ operator, relying on the reader to understand by context which particular substitution operator is intended. The definition of simultaneous substitution for assertion language expressions appears in Tables 10.1, 10.2, and 10.3.

$$\begin{array}{lcl}
n \lhd_v ss & = & n \\
x \lhd_v ss & = & ss\ x \\
(v_1 + v_2) \lhd_v ss & = & (v_1 \lhd_v ss) + (v_2 \lhd_v ss) \\
(v_1 - v_2) \lhd_v ss & = & (v_1 \lhd_v ss) - (v_2 \lhd_v ss) \\
(v_1 * v_2) \lhd_v ss & = & (v_1 \lhd_v ss) * (v_2 \lhd_v ss)
\end{array}$$

Table 10.1: Assertion Numeric Expression Simultaneous Substitution.

$$\begin{array}{lcl}
\langle\,\rangle \lhd_{vs} ss & = & \langle\,\rangle \\
(CONS\ v\ vs) \lhd_{vs} ss & = & CONS\ (v \lhd_v ss)\ (vs \lhd_{vs} ss)
\end{array}$$

Table 10.2: Assertion Numeric Expression List Simultaneous Substitution.

Finally, there is a dual notion of applying a simultaneous substitution to a state, instead of to an expression; this is called *semantic substitution,* and is defined as

$$s \lhd_s ss = \lambda y.\ (V\ (ss\ y)\ s).$$

$$
\begin{array}{lcl}
\mathbf{true} \lhd_a ss & = & \mathrm{T} \\
\mathbf{false} \lhd_a ss & = & \mathrm{F} \\
(v_1 = v_2) \lhd_a ss & = & (v_1 \lhd_v ss) = (v_2 \lhd_v ss) \\
(v_1 < v_2) \lhd_a ss & = & (v_1 \lhd_v ss) < (v_2 \lhd_v ss) \\
(vs_1 \ll vs_2) \lhd_a ss & = & (vs_1 \lhd_{vs} ss) \ll (vs_2 \lhd_{vs} ss) \\
(a_1 \wedge a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \wedge (a_2 \lhd_a ss) \\
(a_1 \vee a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \vee (a_2 \lhd_a ss) \\
(\sim a) \lhd_a ss & = & \sim(a \lhd_a ss) \\
(a_1 \Rightarrow a_2) \lhd_a ss & = & (a_1 \lhd_a ss) \Rightarrow (a_2 \lhd_a ss) \\
(a_1 = a_2) \lhd_a ss & = & (a_1 \lhd_a ss) = (a_2 \lhd_a ss) \\
(a_1 \Rightarrow a_2 \mid a_3) \lhd_a ss & = & (a_1 \lhd_a ss) \Rightarrow (a_2 \lhd_a ss) \mid (a_3 \lhd_a ss) \\
(\mathbf{close}\ a) \lhd_a ss & = & \mathbf{close}\ a \\
 & & \\
(\forall x.\ a) \lhd_a ss & = & \mathbf{let}\ free = \displaystyle\bigcup_{z\ \in\ (FV_a\ a)-\{x\}} FV_v\ (ss\ z)\ \ \mathbf{in} \\
 & & \mathbf{let}\ y = variant\ x\ free\ \mathbf{in} \\
 & & \forall y.\ a \lhd_a (ss[(AVAR\ y)/x]) \\
 & & \\
(\exists x.\ a) \lhd_a ss & = & \mathbf{let}\ free = \displaystyle\bigcup_{z\ \in\ (FV_a\ a)-\{x\}} FV_v\ (ss\ z)\ \ \mathbf{in} \\
 & & \mathbf{let}\ y = variant\ x\ free\ \mathbf{in} \\
 & & \exists y.\ a \lhd_a (ss[(AVAR\ y)/x])
\end{array}
$$

Table 10.3: Assertion Boolean Expression Simultaneous Substitution.

Most of the cases of the definition of the application of a substitution to an expression are simply the distribution of the substitution over the immediate subexpressions. For example, the application of a substitution to a conjunction is

$$(a_1 \wedge a_2) \lhd_a ss \;=\; (a_1 \lhd_a ss) \wedge (a_2 \lhd_a ss)$$

The interesting cases of the definition of $a \lhd_a ss$ are where a is a quantified expression, e.g.:

$$\begin{array}{lcl} (\forall x.\ a) \lhd_a ss & = & \textbf{let}\ free = \bigcup_{z\ \in\ (FV_a\ a) - \{x\}} FV_v\ (ss\ z)\ \ \textbf{in} \\ & & \textbf{let}\ y = variant\ x\ free\ \textbf{in} \\ & & \forall y.\ a \lhd_a (ss[(AVAR\ y)/x]) \end{array}$$

Here FV_v is a function that returns the set of free variables in a numeric assertion expression, FV_a is a function that returns the set of free variables in a boolean assertion expression, and $variant\ x\ free$ is a function that yields a new variable as a variant of x, guaranteed not to be in the set $free$.

Once we have defined substitution as a syntactic manipulation, we can then prove the three theorems in Table 10.4 about the semantics of substitution.

$\vdash \forall v\ s\ ss.$	$V\ (v \lhd_v ss)\ s$	$=$	$V\ v\ (s \lhd_s ss)$
$\vdash \forall vs\ s\ ss.$	$VS\ (vs \lhd_{vs} ss)\ s$	$=$	$VS\ vs\ (s \lhd_s ss)$
$\vdash \forall a\ s\ ss.$	$A\ (a \lhd_a ss)\ s$	$=$	$A\ a\ (s \lhd_s ss)$

Table 10.4: Assertion Language Substitution Lemmas.

This is our statement of the Substitution Lemma of logic, and essentially says that syntactic substitution is equivalent to semantic substitution.

10.2.2 Variables-for-Variables Substitution

The substitutions discussed above replaced variables by (possibly large) numeric expressions. There is a potentially simpler version of substitution, which only replaces variables by variables. We represent these substitutions by functions of type `vsubst` = `var` $\rightarrow$ `var`.

The application of these substitutions to assertion expressions is defined in Tables 10.5, 10.6, and 10.7, defining decorated versions of the operator $\triangleleft$, like the simultaneous substitution described above, but where ss is of type `vsubst`.

$$
\begin{array}{lcl}
n \triangleleft_{vv} ss & = & n \\
x \triangleleft_{vv} ss & = & AVAR\ (ss\ x) \\
(v_1 + v_2) \triangleleft_{vv} ss & = & (v_1 \triangleleft_{vv} ss) + (v_2 \triangleleft_{vv} ss) \\
(v_1 - v_2) \triangleleft_{vv} ss & = & (v_1 \triangleleft_{vv} ss) - (v_2 \triangleleft_{vv} ss) \\
(v_1 * v_2) \triangleleft_{vv} ss & = & (v_1 \triangleleft_{vv} ss) * (v_2 \triangleleft_{vv} ss)
\end{array}
$$

Table 10.5: Assertion Numeric Expression Variable-for-Variable Substitution.

$$
\begin{array}{lcl}
\langle\,\rangle \triangleleft_{vsv} ss & = & \langle\,\rangle \\
(CONS\ v\ vs) \triangleleft_{vsv} ss & = & CONS\ (v \triangleleft_{vv} ss)\ (vs \triangleleft_{vsv} ss)
\end{array}
$$

Table 10.6: Assertion Numeric Expression List Variable-for-Variable Substitution.

Most of the cases are the distribution of the substitution over the immediate subexpressions, as before. The application of ss to an assertion expression which is a simple variable is different, in that applying ss as a function to the variable name x will yield another variable, which then must be converted into an assertion expression using $AVAR$.

$$
\begin{array}{lcl}
\textbf{true} \lhd_{av} ss & = & \mathrm{T} \\
\textbf{false} \lhd_{av} ss & = & \mathrm{F} \\
(v_1 = v_2) \lhd_{av} ss & = & (v_1 \lhd_{vv} ss) = (v_2 \lhd_{vv} ss) \\
(v_1 < v_2) \lhd_{av} ss & = & (v_1 \lhd_{vv} ss) < (v_2 \lhd_{vv} ss) \\
(vs_1 \ll vs_2) \lhd_{av} ss & = & (vs_1 \lhd_{vsv} ss) \ll (vs_2 \lhd_{vsv} ss) \\
(a_1 \wedge a_2) \lhd_{av} ss & = & (a_1 \lhd_{av} ss) \wedge (a_2 \lhd_{av} ss) \\
(a_1 \vee a_2) \lhd_{av} ss & = & (a_1 \lhd_{av} ss) \vee (a_2 \lhd_{av} ss) \\
(\sim a) \lhd_{av} ss & = & \sim(a \lhd_{av} ss) \\
(a_1 \Rightarrow a_2) \lhd_{av} ss & = & (a_1 \lhd_{av} ss) \Rightarrow (a_2 \lhd_{av} ss) \\
(a_1 = a_2) \lhd_{av} ss & = & (a_1 \lhd_{av} ss) = (a_2 \lhd_{av} ss) \\
(a_1 => a_2 \mid a_3) \lhd_{av} ss & = & (a_1 \lhd_{av} ss) => (a_2 \lhd_{av} ss) \mid (a_3 \lhd_{av} ss) \\
(\textbf{close}\ a) \lhd_{av} ss & = & \textbf{close}\ a \\
(\forall x.\ a) \lhd_{av} ss & = & \forall(ss\ x).\ (a \lhd_{av} ss) \\
(\exists x.\ a) \lhd_{av} ss & = & \exists(ss\ x).\ (a \lhd_{av} ss)
\end{array}
$$

Table 10.7: Assertion Boolean Expression Variable-for-Variable Substitution.

More markedly, the cases for the substitution on quantified expressions has greatly simplified, for example

$$(\forall x.\ a) \lhd_{av} ss \;=\; \forall(ss\ x).\ (a \lhd_{av} ss).$$

There is no need here for the avoidance of capture and the selection of new variables, as the bound variable itself is also substituted, which was impossible before.

The most common variable substitutions we will use will replace the variables in one list by those of another list of equal length. We will use ι_v to denote the identity function between variables, ι_v: $\texttt{var} \rightarrow \texttt{var}$. Then we define the operator $//_v$ to construct these variables-for-variables substitutions in Table 10.8. In the rest of this document, we will simply use a single slash to indicate this substitution-creating operator, as in $[ys/xs]$, relying on the reader to realize from the context that since ys and xs are lists of variables, that we are referring to the variables-for-variables substitution creation operator.

$$
\begin{array}{rcl}
[\,[]\,//_v\; xs\,] & = & \iota_v \\
[\,ys\,//_v\,[]\,] & = & \iota_v \\
[\,CONS\; y\; ys\,//_v\; CONS\; x\; xs\,] & = & \mathbf{let}\; ss = [ys\,//_v\; xs]\; \mathbf{in} \\
 & & ss\; [(ss\; x)/(@z.\,(ss\; z) = y)]\; [y/x]
\end{array}
$$

Table 10.8: Variables-for-Variables Substitution Creation operator $//_v$.

In this definition of $//_v$, the `vsubst` ss, which is a mapping from variables to variables, is updated, first binding $(@z.\,(ss\; z) = y)$ to $ss\; x$, and then x to y. @ here is the Hilbert selection operator, choosing and yielding some variable z such that $ss\; z = y$. The reason for this double binding, rather than simply binding x to y, is to preserve the one-to-one property of the mapping; for every variable, there is exactly one variable that maps to it. This makes each such substitution one-to-one, onto, and invertible.

Once we have defined the application of variable-for-variable substitutions as a syntactic manipulation, we can then prove the theorems in Table 10.9 about the semantics of substitution.

These are only some of the shortest and simplest of the theorems proven about this kind of substitution. The ones shown describe the relationship between this kind of substitution, and the previous, where the previous kind is used to apply substitutions of the form $AVAR \circ ss$, which have type `var` $\rightarrow$ `vexp`. There are also three theorems about composing these variable-to-variable substitutions.

$$\vdash \forall v\ ss.\ \ v \lhd_{vv} ss \ = \ v \lhd_{v} (AVAR \circ ss)$$

$$\vdash \forall vs\ ss.\ \ vs \lhd_{vsv} ss \ = \ vs \lhd_{vs} (AVAR \circ ss)$$

$$\vdash \forall ss\ s.\ \ s \lhd_{s} (AVAR \circ ss) \ = \ s \circ ss$$

$$\vdash \forall v\ ss\ s.\ \ V\ (v \lhd_{vv} ss)\ s \ = \ V\ v\ (s \circ ss)$$

$$\vdash \forall vs\ ss\ s.\ \ VS\ (vs \lhd_{vsv} ss)\ s \ = \ VS\ vs\ (s \circ ss)$$

$$\vdash \forall a\ ss\ s.\ \ ONE\_ONE\ ss \Rightarrow (A\ (a \lhd_{av} ss)\ s \ = \ A\ a\ (s \circ ss))$$

$$\vdash \forall a\ ys\ xs\ s.\ \ A\ (a \lhd_{av} [ys/xs])\ s \ = \ A\ a\ (s \circ [ys/xs])$$

$$\vdash \forall a\ ys\ xs\ s.\ \ A\ (a \lhd_{av} [ys/xs])\ s \ = \ A\ (a \lhd_{a} (AVAR \circ [ys/xs])\ s$$

$$\vdash \forall v\ ss_1\ ss_2.\ \ v \lhd_{vv} (ss_2 \circ ss_1) \ = \ (v \lhd_{vv} ss_1) \lhd_{vv} ss_2$$

$$\vdash \forall vs\ ss_1\ ss_2.\ \ vs \lhd_{vsv} (ss_2 \circ ss_1) \ = \ (vs \lhd_{vsv} ss_1) \lhd_{vsv} ss_2$$

$$\vdash \forall a\ ss_1\ ss_2.\ \ a \lhd_{av} (ss_2 \circ ss_1) \ = \ (a \lhd_{av} ss_1) \lhd_{av} ss_2$$

Table 10.9: Assertion Language Var-for-Var Substitution Lemmas.

10.2.3 Programming Language Substitution

If we wish to perform substitutions on *programming language* phrases, instead of assertion language phrases, we run into the difficulty that since expressions can have side effects, it is no longer immaterial how often an expression is evaluated. Hence it is not feasible to consider substitutions where expressions are substituted for variables. However, it turns out that the places where substitutions need to be performed on programming language phrases only require the substitution of variables for variables. Hence we only need to define one set of substitution operators for the Sunrise programming language.

In the following Tables 10.10 through 10.15, we define substitution on lists of variables, numeric expressions, lists of numeric expressions, boolean expressions, commands, and even on progress environments (i.e., *calls*).

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{xs} ss & = & \langle\,\rangle \\
(CONS\ x\ xs) \lhd_{xs} ss & = & CONS\ (ss\ x)\ (xs \lhd_{xs} ss)
\end{array}
$$

Table 10.10: Program Variable List Substitution.

$$
\begin{array}{lcl}
n \lhd_e ss & = & n \\
x \lhd_e ss & = & PVAR\ (ss\ x) \\
(\texttt{++}x) \lhd_e ss & = & \texttt{++}(ss\ x) \\
(e_1 + e_2) \lhd_e ss & = & (e_1 \lhd_e ss) + (e_2 \lhd_e ss) \\
(e_1 - e_2) \lhd_e ss & = & (e_1 \lhd_e ss) - (e_2 \lhd_e ss) \\
(e_1 * e_2) \lhd_e ss & = & (e_1 \lhd_e ss) * (e_2 \lhd_e ss)
\end{array}
$$

Table 10.11: Program Numeric Expression Substitution.

$$
\begin{array}{lcl}
\langle\,\rangle \lhd_{es} ss & = & \langle\,\rangle \\
(CONS\ e\ es) \lhd_{es} ss & = & CONS\ (e \lhd_e ss)\ (es \lhd_{es} ss)
\end{array}
$$

Table 10.12: Program Numeric Expression List Substitution.

$$
\begin{array}{lcl}
(e_1 = e_2) \lhd_b ss & = & (e_1 \lhd_e ss) = (e_2 \lhd_e ss) \\
(e_1 < e_2) \lhd_b ss & = & (e_1 \lhd_e ss) < (e_2 \lhd_e ss) \\
(es_1 \ll es_2) \lhd_b ss & = & (es_1 \lhd_{es} ss) \ll (es_2 \lhd_{es} ss) \\
(b_1 \wedge b_2) \lhd_b ss & = & (b_1 \lhd_b ss) \wedge (b_2 \lhd_b ss) \\
(b_1 \vee b_2) \lhd_b ss & = & (b_1 \lhd_b ss) \vee (b_2 \lhd_b ss) \\
(\sim b) \lhd_b ss & = & \sim(b \lhd_b ss)
\end{array}
$$

Table 10.13: Program Boolean Expression Substitution.

$$
\begin{array}{lcl}
\mathbf{skip} \lhd_c ss & = & \mathbf{skip} \\[1ex]
\mathbf{abort} \lhd_c ss & = & \mathbf{abort} \\[1ex]
(x := e) \lhd_c ss & = & (ss\ x) := (e \lhd_e ss) \\[1ex]
(c_1\ ;\ c_2) \lhd_c ss & = & (c_1 \lhd_c ss)\ ;\ (c_2 \lhd_c ss) \\[1ex]
\begin{pmatrix} \mathbf{if}\ b\ \mathbf{then}\ c_1 \\ \quad \mathbf{else}\ c_2\ \mathbf{fi} \end{pmatrix} \lhd_c ss & = & \begin{array}{l} \mathbf{if}\ (b \lhd_b ss)\ \mathbf{then}\ (c_1 \lhd_c ss) \\ \qquad\qquad \mathbf{else}\ (c_2 \lhd_c ss)\ \mathbf{fi} \end{array} \\[1ex]
\begin{pmatrix} \mathbf{assert}\ a\ \mathbf{with}\ a_{pr} \\ \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od} \end{pmatrix} \lhd_c ss & = & \begin{array}{l} \mathbf{assert}\ (a \lhd_{av} ss)\ \mathbf{with}\ (a_{pr} \lhd_{av} ss) \\ \mathbf{while}\ (b \lhd_b ss)\ \mathbf{do}\ (c \lhd_c ss)\ \mathbf{od} \end{array} \\[1ex]
(\mathbf{call}\ p\ (xs; es)) \lhd_c ss & = & \mathbf{call}\ p\ ((xs \lhd_{xs} ss)\ ;\ (es \lhd_{es} ss))
\end{array}
$$

Table 10.14: Program Command Substitution.

$$g \lhd_g ss \;=\; (\lambda p.\ (g\ p) \lhd_{av} ss)$$

Table 10.15: Program Progress Environment Substitution.

Table 10.16 has programming language versions of the Substitution Lemma.

$$\vdash \forall e\ s_1\ n\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(E\ (e \lhd_e ss)\ s_1\ n\ s_2 \;=\; E\ e\ (s_1 \circ ss)\ n\ (s_2 \circ ss))$$

$$\vdash \forall e\ s_1\ n\ s_2\ ys\ xs.$$
$$E\ (e \lhd_e [ys/xs])\ s_1\ n\ s_2 \;=\; E\ e\ (s_1 \circ [ys/xs])\ n\ (s_2 \circ [ys/xs])$$

$$\vdash \forall es\ s_1\ ns\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(ES\ (es \lhd_{es} ss)\ s_1\ ns\ s_2 \;=\; ES\ es\ (s_1 \circ ss)\ ns\ (s_2 \circ ss))$$

$$\vdash \forall es\ s_1\ ns\ s_2\ ys\ xs.$$
$$ES\ (es \lhd_{es} [ys/xs])\ s_1\ ns\ s_2 \;=\; ES\ es\ (s_1 \circ [ys/xs])\ ns\ (s_2 \circ [ys/xs])$$

$$\vdash \forall b\ s_1\ t\ s_2\ ss.\ \ ONE\_ONE\ ss \wedge ONTO\ ss \Rightarrow$$
$$(B\ (b \lhd_b ss)\ s_1\ t\ s_2 \;=\; B\ b\ (s_1 \circ ss)\ t\ (s_2 \circ ss))$$

$$\vdash \forall b\ s_1\ t\ s_2\ ys\ xs.$$
$$B\ (b \lhd_b [ys/xs])\ s_1\ t\ s_2 \;=\; B\ b\ (s_1 \circ [ys/xs])\ t\ (s_2 \circ [ys/xs])$$

$$\vdash \forall c\ g\ \rho\ ss\ s_1\ s_2.$$
$$WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ ss\ \Rightarrow$$
$$(C\ (c \lhd_c ss)\ \rho\ s_1\ s_2 \;=\; C\ c\ \rho\ (s_1 \circ ss)\ (s_2 \circ ss))$$

$$\vdash \forall c\ g\ \rho\ ys\ xs\ s_1\ s_2.$$
$$WF_{env\_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ [ys/xs]\ \Rightarrow$$
$$(C\ (c \lhd_c [ys/xs])\ \rho\ s_1\ s_2 \;=\; C\ c\ \rho\ (s_1 \circ [ys/xs])\ (s_2 \circ [ys/xs]))$$

Table 10.16: Programming Language Substitution Lemmas.

Finally, we exhibit some theorems that declare that if the free variables of an expression are mapped to the same results by two different variable-for-variable substitutions, then the result of applying the two substitutions to the expression must be the same. Thus the results of substitution depend only on the substitution's effect on the expression's free variables.

$$
\begin{array}{l}
\vdash \forall e\ ss_1\ ss_2.\ \ (\forall x.\ x \in FV_e\ e \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow \\
\qquad (e \lhd_e ss_1 \ = \ e \lhd_e ss_2) \\[1em]
\vdash \forall es\ ss_1\ ss_2.\ \ (\forall x.\ x \in FV_{es}\ es \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow \\
\qquad (es \lhd_{es} ss_1 \ = \ es \lhd_{es} ss_2) \\[1em]
\vdash \forall b\ ss_1\ ss_2.\ \ (\forall x.\ x \in FV_b\ b \Rightarrow (ss_1\ x = ss_2\ x)) \Rightarrow \\
\qquad (b \lhd_b ss_1 \ = \ b \lhd_b ss_2)
\end{array}
$$

Table 10.17: Programming Language Substitution Equality Theorems.

10.3 Translation

Expressions have typically not been trated in previous work on verification; there are some exceptions, notably Sokołowski [Sok84]. Expressions with side effects have been particularly excluded. Since expressions did not have side effects they were often considered to be a sublanguage, common to both the programming language and the assertion language. Thus one would see expressions such as $p \wedge b$, where p was an assertion and b was a boolean expression from the programming language.

One of the key realizations of this work was the need to carefully distinguish these two languages, and not confuse their expression sublanguages. This then requires us to *translate* programming language expressions into equivalent expressions in the assertion language before the two may be combined as above. In fact, since we allow expressions to have side effects, there are actually two results of translating a programming language expression e:

- an assertion language expression, representing the value of e in the state "before" evaluation, *and*
- a simultaneous substitution, representing the change in state from "before" evaluating e to "after" evaluating e.

For example, the translator for numeric expressions is defined using a helper function $VE1$: $\texttt{exp} \rightarrow \texttt{subst} \rightarrow (\texttt{aexp} \times \texttt{subst})$:

$$
\begin{aligned}
VE1\ (n)\ ss &= n,\ ss \\
VE1\ (x)\ ss &= ss\ x,\ ss
\end{aligned}
$$

$$
\begin{aligned}
VE1\ (\text{++}x)\ ss &= (ss\ x) + 1,\ ss[((ss\ x) + 1)/x] \\
VE1\ (e_1 + e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\
&\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 + v_2,\ ss_2)))\ ss \\
VE1\ (e_1 - e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\
&\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 - v_2,\ ss_2)))\ ss \\
VE1\ (e_1 * e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\
&\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 * v_2,\ ss_2)))\ ss
\end{aligned}
$$

where $\rightarrow$ is a "translator continuation" operator, defined as

$$(f \rightarrow k)\ ss = \mathbf{let}\ (v, ss') = f\ ss\ \mathbf{in}\ k\ v\ ss'$$

Then define

$$
\begin{aligned}
VE\ e &= FST\ (VE1\ e\ \iota) \\
VE\_state\ e &= SND\ (VE1\ e\ \iota)
\end{aligned}
$$

where ι is the identity substitution, $\iota\ x = AVAR\ x$. These two functions deliver the two results itemized above for the translation of e.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$
\begin{aligned}
\vdash \forall e\ s_1\ n\ s_2.\ (E\ e\ s_1\ n\ s_2) &= (n = V\ (VE\ e)\ s_1 \wedge \\
&\qquad s_2 = s_1 \lhd (VE\_state\ e))
\end{aligned}
$$

In a similar fashion we can translate lists of numeric expressions. The translator for lists of numeric expressions is defined using a helper function $VES1$: $(\texttt{exp})\texttt{list} \rightarrow \texttt{subst} \rightarrow ((\texttt{aexp})\texttt{list} \times \texttt{subst})$:

$$
\begin{aligned}
VES1\ (\langle\rangle)\ ss &= [],\ ss \\
VE1\ (CONS\ e\ es)\ ss &= (VE1\ e \rightarrow \lambda v. \\
&\quad (VES1\ es \rightarrow \lambda vs\ ss_2.\ (CONS\ v\ vs,\ ss_2)))\ ss
\end{aligned}
$$

Then define

$$\begin{aligned} VES\ es &= FST\ (VES1\ es\ \iota) \\ VES\_state\ es &= SND\ (VES1\ es\ \iota) \end{aligned}$$

These two functions deliver the two results itemized above for the translation of es.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$\begin{aligned} \vdash \forall es\ s_1\ ns\ s_2.\ (ES\ es\ s_1\ ns\ s_2) \;=\; & (ns = VS\ (VES\ es)\ s_1\ \wedge \\ & s_2 = s_1 \lhd (VES\_state\ es)) \end{aligned}$$

In a similar fashion we can translate boolean expressions. The translator for boolean expressions is defined using a helper function

$AB1$: `bexp` $\rightarrow$ `subst` $\rightarrow$ (`aexp` $\times$ `subst`):

$$\begin{aligned} AB1\ (e_1 = e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\ &\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 = v_2,\ ss_2)))\ ss \\ AB1\ (e_1 < e_2)\ ss &= (VE1\ e_1 \rightarrow \lambda v_1. \\ &\quad (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 < v_2,\ ss_2)))\ ss \\ AB1\ (es_1 \ll es_2)\ ss &= (VES1\ es_1 \rightarrow \lambda vs_1. \\ &\quad (VES1\ es_2 \rightarrow \lambda vs_2\ ss_2.\ (vs_1 \ll vs_2,\ ss_2)))\ ss \\ AB1\ (b_1 \wedge b_2)\ ss &= (AB1\ b_1 \rightarrow \lambda t_1. \\ &\quad (AB1\ b_2 \rightarrow \lambda t_2\ ss_2.\ (t_1 \wedge t_2,\ ss_2)))\ ss \\ AB1\ (b_1 \vee b_2)\ ss &= (AB1\ b_1 \rightarrow \lambda t_1. \\ &\quad (AB1\ b_2 \rightarrow \lambda t_2\ ss_2.\ (t_1 \vee t_2,\ ss_2)))\ ss \\ AB1\ (\sim b)\ ss &= (AB1\ b \rightarrow \lambda t\ ss_2.\ (\sim t,\ ss_2))\ ss \end{aligned}$$

Then define

$$\begin{aligned} AB\ b &= FST\ (AB1\ b\ \iota) \\ AB\_state\ b &= SND\ (AB1\ b\ \iota) \end{aligned}$$

These two functions deliver the two results itemized above for the translation of b.

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem.

$$\begin{array}{rcl} \vdash \forall b\ s_1\ t\ s_2.\ (B\ b\ s_1\ t\ s_2) & = & (t = A\ (AB\ b)\ s_1\ \wedge \\ & & \ s_2 = s_1 \lhd (AB\_state\ b)) \end{array}$$

This theorem, along with the corresponding ones for numeric expressions and lists of numeric expressions, mean that every evaluation of a programming language expresssion has its semantics completely captured by the two translation functions for its type. These are essentially small compiler correctness proofs.

Using these translation functions, we may define functions to compute the appropriate preconditions to an executable expression, given the postcondition, as given in Table 10.18.

`vexp`	$ve\_pre\ e\ v = v \lhd_v (VE\_state\ e)$ $ves\_pre\ es\ v = v \lhd_v (VES\_state\ es)$ $vb\_pre\ b\ v = v \lhd_v (AB\_state\ b)$
`(vexp)list`	$vse\_pre\ e\ vs = vs \lhd_{vs} (VE\_state\ e)$ $vses\_pre\ es\ vs = vs \lhd_{vs} (VES\_state\ es)$ $vsb\_pre\ b\ vs = vs \lhd_{vs} (AB\_state\ b)$
`aexp`	$ae\_pre\ e\ a = a \lhd_a (VE\_state\ e)$ $aes\_pre\ es\ a = a \lhd_a (VES\_state\ es)$ $ab\_pre\ b\ a = a \lhd_a (AB\_state\ b)$

Table 10.18: Expression Precondition Functions.

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

$$[x := e] = (VE\_state\ e)[(VE\ e)/x]$$

and

$$[xs := es] = (VES\_state\ es)[(VES\ es)/xs].$$

These simultaneous substitutions are used directly in defining the VCG function. The single assignment substitution is used in processing the assignment command, to compute the appropriate precondition. The multiple assignment substitution is used in processing the actual value parameters of the procedure call command, to reflect their execution's effect on the state.

We have found these translation functions to greatly condense and simplify the handling of expressions with side effects. While not an approach that can describe all possible operators with side effects, we believe this translation function approach is flexible enough to handle input/output and user-defined functions with side effects. These questions are a part of our plans for future research.

10.4 Well-Formedness

In the creation of small languages with simple features, it may be possible to define the semantics of the language sufficiently cleanly so that every program which is syntactically valid has a well-defined and proper semantics. However, as more sophisticated features are added to the language under consideration, it becomes necessary to further restrict the set of "acceptable" programs for which one's analysis is applicable. We have found that the feature of procedure calls introduced the need to verify several restrictions on sample programs, for example that the arity of a call matched that of the definition. We have defined predicates to express these restrictions, called *well-formedness* predicates. Unless a program meets these criteria, we do not even consider it in a proof of correctness.

These well-formedness predicates describe a number of conditions, mostly simple syntactic checks like the arity check mentioned, but also including a number of semantic checks, such as the total correctness of a procedure's body with respect to its precondition and postcondition. Generally, the syntactic checks may be decided by a single, static, compile-time examination of the program. The semantic checks are satisfied by the meta-level verification of the verification condition generator, and by the proofs of the verification conditions generated by it. Since this verification includes some of the hardest parts of the proof of well-formedness, it is fortunate that much of it can be decided at the meta level. For this version of the Sunrise language, we find it unnecessary to also include dynamic checks, to be conjoined to the preconditions computed during the VCG calculation, along with the static checks instituted for compile-time. This may change in the future, for example with the introduction of arrays the checks to prevent aliasing of parameters may require dynamic checks. But for now, the only checks necessary are static, syntactic checks, that may be performed fully automatically.

It is interesting to us that there has been very little focus in the past on this issue of well-formedness. In our work, it became crucial from the beginning of the work on procedures, because it was not possible to properly relate the operational semantics and the axiomatic semantics without constraining the set of programs considered to ones that made sense. We hope that this work will exhibit the issues involved with their proper priority.

10.4.1 Informal Description

The checks that are part of well-formedness vary with the construct being analyzed, but for the most part are simple syntactic tests on the immediate constituent constructs, and so may be defined on the structure of the constructs.

One pervasive check is the exclusion of logical variables from normal program text. Logical variables are restricted from appearing except in assertion language expressions, as part of the definition of the Sunrise language. Yet syntactically, a logical variable and a program variable are both the same kind of phrase. We rely on well-formedness checks to ensure that only program variables appear in normal program text.

For procedure calls, other checks are needed as well. The arity checks are one example, that the number of actual variable parameters matches the number of formal variable parameters, and the same for value parameters. In addition, we must ensure that aliasing does not occur; this can be done by checking that the combination of the actual variable parameters and the declared globals of the procedure being called contains no duplicates.

When it comes to procedure definitions, there are several checks that must be satisfied. These express both syntactic and semantic considerations. The syntactic considerations include checking that every variable in the parameter lists or the globals is not logical, that they have no duplicates among them, that the body of the procedure is well-formed, and constraints on the free variables of the precondition and postcondition.

A procedure definition is fully well-formed if it is syntactically well-formed as described above, and then satisfies one additional semantic criterion; the body must be totally correct with respect to the given precondition and postcondition.

It now becomes possible to speak of an entire environment of procedure definitons being well-formed, if every individual procedure definition in the environment is itself fully well-formed.

The requirement for total correctness is quite strong. It turns out that it is quite useful to establish "stepping stones" along the way to proving total correctness, where less powerful semantic properties are established and then combined to justify more powerful ones. We have made considerable use of this approach, and in the verification of the VCG, we prove several properties in order, as described in Table 10.19.

$WF_{env\_syntax}\ \rho$	ρ is well-formed for syntax
$WF_{envk\_partial}\ \rho\ k$	ρ is well-formed for partial correctness to stage k
$WF_{envk}\ \rho\ k$	ρ is well-formed for syntax and partial correct. to stage k
$WF_{envp}\ \rho$	ρ is well-formed for syntax and 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
$WF_{env\_rec}\ \rho$	ρ is well-formed for recursion
$WF_{env\_partial}\ \rho$	ρ is well-formed for partial correctness
$WF_{env\_total}\ \rho$	ρ is well-formed for termination
$WF_{env\_correct}\ \rho$	ρ is well-formed for total correctness
$WF_{env}\ \rho$	ρ is well-formed for syntax and total correctness

Table 10.19: Procedure Environment Well-Formedness Predicates.

10.4.2 Well-Formedness Predicate Definitions

A string s is well-formed ($WF_s\ s$) if the first character is not '^'.

$$
\begin{array}{|l|}
\hline \\
WF_s \text{ `` } = \mathrm{T} \\
WF_s\ (STRING\ a\ s) = (a \neq LOG\_CHAR) \\
\\
\text{where } LOG\_CHAR = \text{`\^{}'}. \\
\\
\hline
\end{array}
$$

Table 10.20: Definition of Well-Formedness for Strings.

A variable x is well-formed ($WF_x\ x$) if its string is well-formed.

$$
\boxed{WF_x\ (VAR\ s\ n) = WF_s\ s}
$$

Table 10.21: Definition of Well-Formedness for Variables.

A list of variables xs is well-formed ($WF_{xs}\ xs$) if every variable in the list is well-formed.

$$
\begin{array}{|l|}
\hline \\
WF_{xs}\ \langle\rangle = \mathrm{T} \\
WF_{xs}\ (CONS\ x\ xs) = WF_x\ x\ \wedge\ WF_{xs}\ xs \\
\\
\hline
\end{array}
$$

Table 10.22: Definition of Well-Formedness for Lists of Variables.

A list of variables xs is not-well-formed ($NOT\_WF_{xs}\ xs$) if every variable in the list is not well-formed.

$$
\begin{array}{|l|}
\hline
\\
NOT\_WF_{xs}\ \langle\,\rangle = \mathrm{T} \\
NOT\_WF_{xs}\ (CONS\ x\ xs) = \sim(WF_x\ x)\ \wedge\ NOT\_WF_{xs}\ xs \\
\\
\hline
\end{array}
$$

Table 10.23: Definition of Not-Well-Formedness for Lists of Variables.

A numeric expression e is well-formed ($WF_e\ e$) if every part is well-formed.

$$
\begin{array}{|l|}
\hline
\\
WF_e\ (n) = \mathrm{T} \\
WF_e\ (x) = WF_x\ (x) \\
WF_e\ (++x) = WF_x\ (x) \\
WF_e\ (e_1 + e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_e\ (e_1 - e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_e\ (e_1 * e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
\\
\hline
\end{array}
$$

Table 10.24: Definition of Well-Formedness for Numeric Expressions.

A list of numeric expressions es is well-formed ($WF_{es}\ es$) if every expression in the list is well-formed.

$$
\begin{array}{|l|}
\hline
\\
WF_{es}\ \langle\,\rangle = \mathrm{T} \\
WF_{es}\ (CONS\ e\ es) = WF_e\ e\ \wedge\ WF_{es}\ es \\
\\
\hline
\end{array}
$$

Table 10.25: Definition of Well-Formedness for Lists of Numeric Expressions.

A boolean expression b is well-formed ($WF_b\ b$) if every part is well-formed.

$$
\begin{array}{l}
WF_b\ (e_1 = e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_b\ (e_1 < e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2 \\
WF_b\ (es_1 \ll es_2) = WF_{es}\ es_1\ \wedge\ WF_{es}\ es_2 \\
WF_b\ (b_1 \wedge b_2) = WF_b\ b_1\ \wedge\ WF_b\ b_2 \\
WF_b\ (b_1 \vee b_2) = WF_b\ b_1\ \wedge\ WF_b\ b_2 \\
WF_b\ (\sim b) = WF_b\ b
\end{array}
$$

Table 10.26: Definition of Well-Formedness for Boolean Expressions.

A command c is well-formed in a progress environment g and a procedure environment ρ ($WF_c\ c\ g\ \rho$) if every part is well-formed, if every **while** command's progress expression avoids other logical variables, if every call supplies the same number of actual parameters as the procedure has formal parameters, and if there is no aliasing among the variable parameters and the globals.

$$
\begin{array}{l}
WF_c\ (\textbf{skip})\ g\ \rho = \mathrm{T} \\
WF_c\ (\textbf{abort})\ g\ \rho = \mathrm{T} \\
WF_c\ (x := e)\ g\ \rho = WF_x\ x\ \wedge\ WF_e\ e \\
WF_c\ (c_1\ ;\ c_2)\ g\ \rho = WF_c\ c_1\ g\ \rho\ \wedge\ WF_c\ c_2\ g\ \rho \\
WF_c\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \textbf{fi})\ g\ \rho = WF_b\ b\ \wedge\ WF_c\ c_1\ g\ \rho\ \wedge\ WF_c\ c_2\ g\ \rho \\
WF_c\ (\textbf{assert}\ a\ \textbf{with}\ a_{pr}\textbf{while}\ b\ \textbf{do}\ c\ \textbf{od})\ g\ \rho = \\
\qquad\qquad WF_b\ b\ \wedge\ WF_c\ c\ g\ \rho\ \wedge \\
\qquad\qquad (\exists v\ x.\ a_{pr} = (v < x)\ \wedge\ \sim(WF_x\ x)\ \wedge\ x \notin (FV_a\ a \cup FV_v\ v)\ \wedge \\
\qquad\qquad\qquad (\forall p.\ x \notin FV_a\ (g\ p)) \\
WF_c\ (\textbf{call}\ p\ (xs; es))\ g\ \rho = \\
\qquad\qquad WF_{xs}\ xs\ \wedge\ WF_{es}\ es\ \wedge \\
\qquad\qquad (|vars| = |xs|)\ \wedge\ (|vals| = |es|)\ \wedge\ DL\ (xs\ \&\ glbs)
\end{array}
$$

Table 10.27: Definition of Well-Formedness for Commands.

where in the last line, $\rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle$. Here $|xs|$

denotes the length of the list xs, and DL ("distinct list") is a predicate saying the variables in xs and $glbs$ have no duplicates.

A procedure specification $\langle vars, vals, glbs, pre, post, calls, rec, c\rangle$ is syntactically well-formed in an environment ρ
($WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$) if

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{let } x = vars \mathbin{\&} vals \mathbin{\&} glbs \textbf{ and } x_0 = logicals\ x \textbf{ in}} \\
 & \\
1) & WF_{xs}\ x \\
2) & DL\ x \\
3) & WF_c\ c\ calls\ \rho \\
4) & GV_c\ c\ \rho \subseteq glbs \\
5) & FV_c\ c\ \rho \subseteq x \\
6) & FV_a\ pre \subseteq x \\
7) & FV_a\ post \subseteq (x \cup x_0) \\
8) & (\forall s.\ \textbf{let } \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle = \rho\ s \textbf{ in} \\
 & \qquad \textbf{let } x' = vars' \mathbin{\&} vals' \mathbin{\&} glbs' \textbf{ in} \\
 & \qquad FV_a\ (calls\ s) \subseteq SL\ (x' \mathbin{\&} x_0)) \\
9) & (rec = \textbf{false}) \vee \\
 & (\exists v\ y.\ rec = (v < y)\ \wedge\ \sim(WF_x\ y)\ \wedge\ FV_v\ v \subseteq SL\ x)
\end{array}
$$

Table 10.28: Definition of Well-Formedness for Procedure Specification Syntax.

The several clauses of the definition of WF_{proc_syntax} are explained as follows:

1. every variable in $vars$, $vals$, and $glbs$ is well-formed, i.e., not logical,
2. the variables in $vars$, $vals$, and $glbs$ have no duplicates,
3. c is well-formed in calls progress environment $calls$ and environment ρ,
4. all globals referenced by procedures called within c are in $glbs$,

5. all the free variables of c are in x,

6. all the free variables of pre are in x,

7. all the free variables of $post$ are in x or in x_0,

8. for each procedure p, all the free variables of the progress expression contained in $calls$ for p are in x' or in x_0, where x' is the list of the variables accessible from p,

9. either rec is **false**, or else rec has the form $v < y$, where y is a logical variable not in x.

A procedure specification $\langle vars, vals, glbs, pre, post, calls, rec, c\rangle$ is fully well-formed (both syntactically and semantically) in an environment ρ ($WF_{proc}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$) if

let $x = vars \,\&\, vals \,\&\, glbs$ **and** $x_0 = logicals\ x$ **in**
1) $WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho$
2) $\{x_0 = x\ \wedge\ pre\}\ c\ \{post\}\ /\rho$

Table 10.29: Definition of Well-Formedness for Procedure Specification.

where

1. the specification is syntactically well-formed, and

2. c is partially correct with precondition $(x_0 = x\ \wedge\ pre)$ and postcondition $post$ in environment ρ.

An environment ρ is well-formed ($WF_{env}\ \rho$) if every procedure declaration is well-formed in ρ.

$$\boxed{WF_{env}\ \rho = \forall p.\ WF_{proc}\ (\rho\ p)\ \rho}$$

Table 10.30: Definition of Well-Formedness for Procedure Environment.

A progress environment *calls* is well-formed in a procedure environment ρ if for every procedure p, all the free variables of $(calls\ p)$ are in the variables accessible from p.

$$\begin{array}{ll} WF_{calls}\ calls\ \rho = (\forall p. & \mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \mathbf{in} \\ & \mathbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \mathbf{in} \\ & (\exists z.\ FV_a\ (calls\ p) \subseteq SL\ (x\ \&\ logicals\ z)) \end{array}$$

Table 10.31: Definition of Well-Formedness for Progress Environment.

A declaration d is well-formed in an environment ρ ($WF_d\ d\ \rho$) if every individual procedure declaration is syntactically well-formed in ρ.

$$\begin{array}{l} WF_d\ (\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c)\ \rho = \\ \qquad\qquad WF_{proc\_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho \\ WF_d\ (d_1\ ;\ d_2)\ \rho = WF_d\ d_1\ \rho\ \wedge\ WF_d\ d_2\ \rho \\ WF_d\ (\mathbf{empty})\ \rho = \mathrm{T} \end{array}$$

Table 10.32: Definition of Well-Formedness for Declarations.

The empty procedure environment ρ_0 is an initial environment with each procedure having no parameters or globals, a **false** precondition and a **true** postcondition, a **false** progress condition for every procedure (none of which are called), a **false** recursion expression, and a body consisting solely of the command **abort**. Declarations present in the program override these default declarations, which should never be invoked. The **false** precondition itself implies the impossibility of proving any program that calls an undeclared procedure.

$$\boxed{\rho_0 \ = \ (\lambda p.\ [\,],\ [\,],\ [\,],\ \mathbf{false},\ \mathbf{true},\ (\lambda p.\ \mathbf{false},)\ \mathbf{false},\ \mathbf{abort})}$$

Table 10.33: Definition of Empty Progress Environment.

The empty progress environment g_0 is **true** for all procedures. This is the progress environment used for processing the main body, for which there are no **calls** ... **with** specifications.

$$\boxed{g_0 \ = \ (\lambda p.\ \mathbf{true})}$$

Table 10.34: Definition of Empty Progress Environment.

A program π is well-formed ($WF_p\ \pi$) if both its declarations and its body are well-formed in the environment the declarations create.

$$
\begin{array}{l}
WF_p\ (\textbf{program}\ d\ ;\ c\ \textbf{end program})\ = \\
\qquad \textbf{let}\ \rho = mkenv\ d\ \rho_0\ \textbf{in} \\
\qquad WF_d\ d\ \rho\ \wedge\ WF_c\ c\ g_0\ \rho
\end{array}
$$

Table 10.35: Definition of Well-Formedness for Programs.

These well-formedness predicates were indispensable prerequisites for all the reasoning of the verification condition generator. They restricted the set of programs considered to those that were consistent and proper. Without these restrictions, no deep theorems about the semantics of the VCG would have been possible; but with them, principles can be stated and proved about the wide class of normal programs which are the actual aim.

10.5 Semantic Stages

When we began proving partial correctness from the conditions generated by the VCG, we ran into a difficulty. Two of the correctness properties we wanted to prove were `vcgcp_THM` and `vcgd_THM`, repeated from Chapter 7 in Table 10.36.

`vcgcp_THM`	$\forall c\ p\ calls\ q\ \rho.\ \ WF_{envp}\ \rho\ \wedge\ WF_c\ c\ calls\ \rho\ \Rightarrow$ **all_el close** $(vcgc\ p\ c\ calls\ q\ \rho) \Rightarrow$ $\{p\}\ c\ \{q\}\ /\rho$
`vcgd_THM`	$\forall d\ \rho.\ \ \rho = mkenv\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ **all_el close** $(vcgd\ d\ \rho) \Rightarrow$ $WF_{envp}\ \rho$

Table 10.36: Repeated VCG verification theorems.

In order to prove `vcgd_THM`, we wished to use `vcgcp_THM`, proven earlier. `vcgd_THM` is used to prove the well-formedness of an environment for partial correctness. This has both syntactic and semantic parts. The syntactic well-formedness is supported by $WF_d\ d\ \rho$, so we need only add the proof of the semantic part. For this, we wished to reason from the truth of the verification conditions produced by $vcgd\ d\ \rho$ to the partial correctness of each procedure body declared in d, with respect to its precondition and postcondition. For this task, `vcgcp_THM` appeared to be the appropriate tool, applying it to each procedure body in turn. The problem was that `vcgcp_THM` itself requires an environment well-formed for partial correctness as a precondition! Thus it seemed to be necessary to know that the environment was well-formed before we could prove that it was well-formed, a circular argument.

The solution was to cut the circle by establishing *stages* of well-formedness for the environment, indexed by number, and to show eventually by numeric induction that all stages hold, and thus the environment is well-formed. Each increase in the index signifies an ability to call procedures to one more level of calling depth. Thus, index 0 designates an environment which is well-formed as long as no procedure calls are made; index 1 designates an environment which is well-formed under calls of procedures which do not themselves issue procedure calls, etc. In pursuing this line of reasoning, it became apparent that in order to define stages of well-formedness, we needed to establish stages of command partial correctness specifications, and of the command semantic relation itself.

The new staged version of the command semantic relation C_k is described in Table 10.37.

$C_k\ c\ \rho\ k\ s_1\ s_2$	command c:`cmd` evaluated in environment ρ and state s_1 yields state s_2, without ever issuing calls beyond a nested depth of k.

Table 10.37: Staged command semantic relation description.

The definition of the new staged command semantic relation C_k is given in Table 10.38. It is similar to the definition of C in Table 5.11, but C_k adds one new argument k, which is the stage number, and every rule maintains that the stage of the resulting tuple is greater than or equal to the stages of all antecedent tuples, *except* for the procedure call rule, where the stage of the result tuple (regarding the procedure call) is exactly one greater than that of the antecedent rule (regarding the procedure's body).

Skip:

$$\frac{}{C_k\ \mathbf{skip}\ \rho\ k\ s\ s}$$

Abort:

(no rules)

Assignment:

$$\frac{E\ e\ s_1\ n\ s_2}{C_k\ (x := e)\ \rho\ k\ s_1\ s_2[n/x]}$$

Sequence:

$$\frac{\begin{array}{l} C_k\ c_1\ \rho\ k_1\ s_1\ s_2, \quad k_1 \leq k \\ C_k\ c_2\ \rho\ k_2\ s_2\ s_3, \quad k_2 \leq k \end{array}}{C_k\ (c_1\ ;\ c_2)\ \rho\ k\ s_1\ s_3}$$

Conditional:

$$\frac{B\ b\ s_1\ \mathrm{T}\ s_2, \qquad C_k\ c_1\ \rho\ k\ s_2\ s_3}{C_k\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ \rho\ k\ s_1\ s_3}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2, \qquad C_k\ c_2\ \rho\ k\ s_2\ s_3}{C_k\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi})\ \rho\ k\ s_1\ s_3}$$

Iteration:

$$\frac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2 \\ C_k\ c\ \rho\ k_1\ s_2\ s_3, \qquad k_1 \leq k \\ C_k\ (\mathbf{assert}\ a\ \mathbf{with}\ pr \\ \quad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ k_2\ s_3\ s_4, \quad k_2 \leq k \end{array}}{\begin{array}{l} C_k\ (\mathbf{assert}\ a\ \mathbf{with}\ pr \\ \quad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ k\ s_1\ s_4 \end{array}}$$

$$\frac{B\ b\ s_1\ \mathrm{F}\ s_2}{\begin{array}{l} C_k\ (\mathbf{assert}\ a\ \mathbf{with}\ pr \\ \quad \mathbf{while}\ b\ \mathbf{do}\ c\ \mathbf{od})\ \rho\ k\ s_1\ s_2 \end{array}}$$

Call:

$$\frac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ vals' = variants\ vals\ (SL\ (xs\ \&\ glbs)) \\ C_k\ (c \lhd [xs\ \&\ vals'/vars\ \&\ vals])\ \rho\ k\ s_2[ns/vals']\ s_3 \end{array}}{C_k\ (\mathbf{call}\ p(xs;\ es))\ \rho\ (k+1)\ s_1\ s_3[(\mathbf{map}\ s_2\ vals')/vals']}$$

Table 10.38: Staged Command Structural Operational Semantics.

We define the staged command partial correctness specification in Table 10.39.

$$\{a_1\}\ c\ \{a_2\}\ /\rho, k \ = \ (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge C_k\ c\ \rho\ k\ s_1\ s_2 \Rightarrow A\ a_2\ s_2)$$

Table 10.39: Staged command Partial Correctness Specification.

We define the staged version of well-formedness of environments for partial correctness in Table 10.40.

$$\begin{array}{l}
WF_{prock}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho\ k \ = \\
\qquad \mathbf{let}\ x = vars\ \&\ vals\ \&\ glbs\ \mathbf{and}\ x_0 = logicals\ x\ \mathbf{in} \\
\qquad (WF_{proc_syntax}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle\ \rho\ \wedge \\
\qquad\ \{x_0 = x\ \wedge\ pre\}\ c\ \{post\}\ /\rho, k) \\
\\
WF_{envk}\ \rho\ k \ = \ \forall p.\ WF_{prock}\ (\rho\ p)\ \rho\ k
\end{array}$$

Table 10.40: Staged Well-Formed Environment Predicate for Partial Correctness.

Using these definitions, we can prove many staged version of previous theorems about commands, for example the substitution lemmas in Table 10.41.

$$\begin{array}{l}
\vdash \forall c\ g\ \rho\ k\ ss\ s_1\ s_2. \\
\qquad WF_{env_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ ss\ \Rightarrow \\
\qquad (C_k\ (c \lhd_c ss)\ \rho\ k\ s_1\ s_2 \ = \ C_k\ c\ \rho\ k\ (s_1 \circ ss)\ (s_2 \circ ss)) \\
\\
\vdash \forall c\ g\ \rho\ k\ ys\ xs\ s_1\ s_2. \\
\qquad WF_{env_syntax}\ \rho\ \wedge\ WF_c\ c\ g\ \rho\ \wedge\ WF_{csubst}\ c\ \rho\ [ys/xs]\ \Rightarrow \\
\qquad (C_k\ (c \lhd_c [ys/xs])\ \rho\ k\ s_1\ s_2 \ = \ C_k\ c\ \rho\ k\ (s_1 \circ [ys/xs])\ (s_2 \circ [ys/xs]))
\end{array}$$

Table 10.41: Staged Command Substitution Lemmas.

We also prove theorems which relate C_k, $\{a_1\}\ c\ \{a_2\}\ /\rho, k$, and $WF_{envk}\ \rho\ k$ to their unstaged original counterparts. These are given in Table 10.42.

$$
\begin{array}{l}
\vdash \forall c\ \rho\ s_1\ s_2. \\
\qquad C\ c\ \rho\ s_1\ s_2 \;=\; (\exists k.\ C_k\ c\ \rho\ k\ s_1\ s_2) \\[1em]
\vdash \forall a_1\ c\ a_2\ \rho. \\
\qquad \{a_1\}\ c\ \{a_2\}\ /\rho \;=\; (\forall k.\ \{a_1\}\ c\ \{a_2\}\ /\rho, k) \\[1em]
\vdash \forall \rho. \\
\qquad WF_{envp}\ \rho \;=\; (\forall k.\ WF_{envk}\ \rho\ k)
\end{array}
$$

Table 10.42: Unstaged-to-Staged Correspondances.

This last theorem gives us the means to prove that an environment is well-formed for partial correctness. We first prove that for $k = 0$, the antecedents of `vcgd_THM` imply the environment is well-formed to stage 0. This is theorem `vcgd_0_THM` of Table 7.3. To prove this, we first prove theorem `vcg1_0_THM` of Table 7.1, and then `vcgc_0_THM` of Table 7.2, from which `vcgd_0_THM` follows.

Then assumming the antecedents of `vcgd_THM` and that the environment is well-formed to stage k, we prove that it is well-formed to stage $k + 1$. This is theorem `vcgd_k_THM`, built as before by first proving `vcg1_k_THM` and then `vcgc_k_THM`. By induction, the environment is then well-formed for all stages, and by the above theorem in Table 10.42, the environment is completely well-formed for partial correctness, which proves theorem `vcgd_THM`.

By proving this induction on stage numbers here at the meta-level, we obviate the need for the programmer to have to prove verification conditions that deal with these partial correctness issues of the program's recursion, for all programs.

CHAPTER 11

Total Correctness

"For He will finish the work and cut it short in righteousness,

Because the LORD will make a short work upon the earth."

— Romans 9:28

"For the Lord GOD of hosts

Will make a determined end

In the midst of all the land."

— Isaiah 10:23

The proof of the termination of programs, and hence their total correctness, is presented in this chapter. We start with the assumptions of partial correctness, precondition maintenance, conditional termination, and most importantly, recursiveness, and prove the termination of every call of every procedure declared in the mutually recursive procedure environment. This leads to an environment which has been verified to be well-formed for total correctness, and thus to be fully well-formed. The total correctness of the environment becomes the last essential element in the proof of the ultimate theorem of this work, Theorem 7.12, as presented in Chapter 7, that the verification condition generator has been verified for total correctness.

Total Correctness has two aspects, partial correctness and termination. In the past these have sometimes been proven apart from each other, and sometimes together, often using the same overall proof structure. But there has begun to appear evidence that there is a more substantial difference between partial correctness and termination than had originally been thought, when recursive procedures are present. In 1990, America and de Boer reported [AdB90]

> ...we may conclude that reasoning about total correctness differs from partial correctness in a substantial way which has not been recognized til now.

In the course of this work, this difference has been exposed and explored. It became evident during the construction of the verification of the VCG that partial correctness was a necessary precursor to even beginning the attack on total correctness. Many of the rules presented in Chapter 6 in the entrance logic and in the termination logic contained partial correctness specifications as necessary antecedents. Moreover, to prove the environment was well-formed for recursion, it was necessary first to have the entire environment established to be well-formed for partial correctness and for calls progress. In a similar way, we will add the assumption that the environment is well-formed for conditional termination, and prove from these that the environment is well-formed for total correctness.

We have already seen a substantial argument was in order to prove the full recursiveness property for procedures, how it was necessary to introduce the entrance logic in order to verify the progress claimed in the *calls* progress expressions in the headers of procedures, and how it was necessary to introduce the analysis of the call graph structure to verify that the progress claimed in the recursion expressions were supported by the progress of the *calls* progress expressions. We also saw it was necessary to introduce the termination logic in order to verify the conditional termination of commands. Now all of these elements come together as necessary precursors to the proof of termination of every procedure. This extended proof, with these layers and stages of development, demonstrates the depth of reasoning that is necessary to prove termination. The good part of this is that once done, it need not be repeated when the VCG is applied. The verification of the VCG allows it to be used without repeating the intricate arguments expressed and proven at the meta level here.

We will begin by summarizing the substance of the argument up to this point.

11.1 Reprise

11.1.1 Entrance Logic

In Section 6.3, we presented an Entrance Logic, including correctness specifications of the forms

$\{a_1\}\ c \to p\ \{a_2\}\ /\rho$	entrance specification
$\{a\}\ c \to \mathbf{pre}\ /\rho$	precondition entrance specification
$\{a\}\ c \to calls\ /\rho$	calls entrance specification
$\{a_1\}\ p_1 - ps \to p_2\ \{a_2\}\ /\rho$	path entrance specification
$\{a_1\}\ p \hookleftarrow \{a_2\}\ /\rho$	recursive entrance specification

We then presented the rules of the Entrance Logic which supported proofs of these correctness specifications for specific program fragments. Later we saw how these rules supported the verification of parts of the VCG, that the truth of the verification conditions produced by the syntax-directed analysis of a procedure's body sufficed to guarantee the partial correctness of the body with respect to the given precondition and postcondition, to guarantee the progress claimed by the *calls* specifications, and to guarantee the achievement of the preconditions of every called procedure at their entrance.

11.1.2 Termination Logic

In Section 6.4, we presented a Termination Logic, including correctness specifications of the forms

$[a]\ c \downarrow / \rho$	command conditional termination specification
$p \downarrow / \rho$	procedure conditional termination specification
$[a]\ c \Downarrow / \rho$	termination specification

We then presented the rules of the Termination Logic which supported proofs of these correctness specifications for specific program fragments. Later we saw how these rules supported the verification of parts of the VCG, that the truth of the verification conditions produced by the syntax-directed analysis of a procedure's body sufficed to guarantee the conditional termination of that body, given the termination of every procedure called immediately from that body.

11.1.3 Recursiveness

Given the properties proven about the environment of all defined procedures, that it was well-formed for partial correctness, precondition maintenance, calls progress, and conditional termination, we showed in Section 7.1.3 a series of functions defined as part of the VCG that analyzed the procedure call graph and produced a list of verification conditions, whose proof, along with the progress claimed by the *calls* specifications previously shown, was sufficient to prove the full recursiveness property, that every recursive call evidenced the progress claimed in the recursion expression for that procedure.

That progress expressed in the form $v < x$, that the value of the expression v strictly decreased from the initial call to the recursive call. This was an example of an expression whose value was a member of a well-founded set, in this case the nonnegative integers. Well-founded sets have the property that there are no infinitely decreasing sequences of values from the set. This lays the foundation for the argument for termination, that if there were a procedure call that exhibited infinite recursive descent, then taking the sequence of values of v at each recursive entrance of the procedure would exhibit such an infinitely decreasing sequence. Since that is excluded by the definition of well-founded sets, there cannot be such a nonterminating procedure call.

11.2 Termination

We will now present the main points of our proof of the termination of mutually recursive procedures. We begin by defining two more semantic relations.

These semantic relations, $terminates$ and $Depth\_calls$, are defined in Tables 11.1 and 11.2. These are related to the semantic relations defined in Chapter 5. First, $terminates$ expresses the condition that a particular procedure's body terminates when started in a given state. Then $Depth\_calls$ connects a procedure name and a state to another procedure name and a state, where there is an execution sequence between the first state at the entrance of the first procedure through nested calls to the second state at the entrance of the second procedure. Of particular interest is that $Depth\_calls$ specifies the length of the chain of calls as a particular integer. Thus $Depth\_calls$ provides a way to describe calls which are nested a particular number of calls deep from the original point where the execution began.

$$
\begin{array}{l}
terminates\ p\ s\ \rho\ = \\
\qquad \textbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p\ \textbf{in} \\
\qquad (\exists s'.\ C\ c\ \rho\ s\ s')
\end{array}
$$

Table 11.1: Termination Semantic Relation $terminates$.

$$
\begin{array}{l}
Depth\_calls\ 0\ p_1\ s_1\ p_2\ s_2\ \rho\ = \\
\qquad p_1 = p_2\ \wedge\ s_1 = s_2 \\
\\
Depth\_calls\ (n+1)\ p_1\ s_1\ p_2\ s_2\ \rho\ = \\
\qquad \exists p_3\ s_3.\ M\_calls\ p_1\ s_1\ [\,]\ p_3\ s_3\ \rho\ \wedge \\
\qquad\qquad Depth\_calls\ n\ p_3\ s_3\ p_2\ s_2\ \rho
\end{array}
$$

Table 11.2: Termination Semantic Relation $Depth\_calls$.

11.2.1 Sketch of Proof

We will first give an sketch of our proof of termination, and then develop that sketch in detail.

SKETCH:

Every command terminates if all of its immediate calls terminate. Hence, it follows that every procedure body terminates if for any n, all of the body's calls at depth n or less terminate. Thus, to show a procedure body terminates, it suffices to show there is an n such that all of the body's calls of depth n or less terminate.

Assume the opposite, that for some procedure body and initial state, that for all n, there is some call at depth n or less which does not terminate. Then there is some call at depth n which does not terminate, for all n. This implies there exists an infinite sequence of nested procedure calls issuing from the original procedure body and state which do not terminate. Consider this sequence of procedures which are called and the states at their entrances. There must be some procedure which occurs an infinite number of times in this sequence, or else the sequence could not itself be infinite, since there is only a finite number of declared procedures. Let p be such a procedure that occurs an infinite number of times, and let v be its recursion expression. Form the infinite sequence of the values of v in the states at every occurrence of p in the first sequence. By the recursiveness property, we have that every pair of values in this sequence is strictly decreasing, and hence this sequence is strictly decreasing. This is then an infinite sequence of decreasing values. But since the set of nonnegative integers is a well-founded set, no such infinite decreasing sequence can exist. Hence our original assumption was wrong, and we may conclude the opposite, that for some n, all of the original procedure's body's calls at depth n or less terminate. As we have shown above, this then implies that the procedure body terminates, unconditionally.

The termination of procedure bodies, combined with the termination of commands based on their immediate calls terminating, gives us that all commands terminate unconditionally. Combining this with the partial correctness of commands gives us the total correctness of commands. The total correctness of commands implies the total correctness of procedure bodies, and hence the entire environment is proved to be fully well-formed.

End of **SKETCH.**

We will now elaborate the sketch.

11.2.2 Termination of Deep Calls

First, we have already shown that every command terminates if all of its immediate calls terminate. That is, consider a command c begun execution in a state s_1. Let s_2 be any possible state which is reachable from s_1 by being the state at the entrance of a procedure called immediately from c. If for all such s_2, the body of that procedure when begun in s_2 terminates, then c must terminate. This last statement is guaranteed by the definition of WF_{env_term}, which is verified to hold based on the syntax-directed part of the VCG and the verification conditions it produces, by the theorem `vcgd_TERM`, given in Table 7.3. It is the primary starting point for the rest of this argument.

Since every command terminates if all of its immediate calls terminate, this also applies to the commands which are the bodies of procedures. Therefore every procedure body terminates if all of its immediate calls terminate. But then consider those immediate calls. Each one of those causes the execution of a procedure body, whose termination is implied by the termination of *its* immediate calls. We may then restate this, that the original procedure body would be guaranteed of terminating if all of the procedure calls at the second level down terminate. More generally, if the original body terminates if all calls at the nth level terminate, then since each one of those calls at the nth level terminates if all their immediate calls terminate, we may say that the original body terminates if all calls at the $(n+1)$th level terminate. Then by induction on n, we say that for any n, if the calls at depth n terminate, then the original body terminates.

In Table 11.3, we have proven that a call at one depth implies that there exist calls at all lesser (more shallow) depths.

$$
\begin{array}{l}
\vdash \forall n\ m\ p_1\ s_1\ p_2\ s_2\ \rho. \\
\qquad WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ p_1))\ s_1\ \wedge \\
\qquad Depth\_calls\ n\ p_1\ s_1\ p_2\ s_2\ \rho\ \wedge \\
\qquad m \leq n \ \Rightarrow \\
\qquad (\exists p_3\ s_3 . Depth\_calls\ m\ p_1\ s_1\ p_3\ s_3\ \rho)
\end{array}
$$

Table 11.3: Theorem of existence of shallower calls.

We have as a theorem in Table 11.4 that if all the calls at one depth or less terminate, then the original procedure call terminates. Since the termination of all the calls at one depth implies the termination of all the calls at one less depth, then by induction we can prove the termination of all calls at shallower depth.

Contrariwise, in Table 11.5 we have proven that if a call at one depth from the original call does not terminate, then for all greater depths, there is a call at that depth from the original call. This is valuable, but it does not yet give us the

$$
\begin{array}{l}
\vdash \forall \rho.\ \ WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho \Rightarrow \\
\qquad (\forall n\ p_1\ s_1.\ (\forall m\ p_2\ s_2.\ m \leq n \wedge Depth\_calls\ m\ p_1\ s_1\ p_2\ s_2\ \rho \Rightarrow \\
\qquad\qquad\qquad\qquad terminates\ p_2\ s_2\ \rho) \wedge \\
\qquad\qquad\qquad (\mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ p_1\ \mathbf{in} \\
\qquad\qquad\qquad\qquad A\ pre\ s_1) \Rightarrow \\
\qquad\qquad\qquad terminates\ p_1\ s_1\ \rho
\end{array}
$$

Table 11.4: Theorem of termination of shallower calls.

existence of an infinite chain of calls, because it does not include the condition that every procedure and state in the chain actually arose from a call from the previous procedure and state.

$$
\begin{array}{l}
\vdash \forall m\ n\ p_1\ s_1\ p_2\ s_2\ \rho. \\
\qquad WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ p_1))\ s_1 \wedge \\
\qquad Depth\_calls\ n\ p_1\ s_1\ p_2\ s_2\ \rho \wedge \\
\qquad \sim(terminates\ p_2\ s_2\ \rho)\ \ \Rightarrow \\
\qquad (\exists p_3\ s_3. Depth\_calls\ (n+m)\ p_1\ s_1\ p_3\ s_3\ \rho\ \ \wedge\ \ \sim(terminates\ p_3\ s_3\ \rho))
\end{array}
$$

Table 11.5: Theorem of existence of all deeper calls.

11.2.3 Existence of an Infinite Sequence

In the termination proof sketch, at one point we assume that there does not exists any n such that all calls of depth n or less terminate. Then for all n there must be some call at depth n or less which does not terminate. This then should imply the existence of an infinite sequence of deeper and deeper calls.

We will prove the existence of such an infinite sequence by actually constructing and exhibiting one. First, we define the function $mk\_sequence$ in Table 11.6 as a generator function to take a pair $\langle p, s\rangle$ of a procedure name and a state, and return the next $\langle p', s'\rangle$ of the infinite sequence.

$$
\begin{array}{l}
mk\_sequence\ 0\ \rho\ p\ s\ =\ p, s \\[1em]
mk\_sequence\ (i+1)\ \rho\ p\ s\ = \\
\qquad \textbf{let}\ (p', s')\ =\ @\ (p', s').\ \ M\_calls\ p\ s\ [\,]\ p'\ s'\ \rho\ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \sim(terminates\ p'\ s'\ \rho) \\
\qquad \textbf{in}\ mk\_sequence\ i\ \rho\ p'\ s'
\end{array}
$$

Table 11.6: Sequence Generator Function $mk\_sequence$.

Here @ is the Hilbert choice operator, which returns some element of its range type which satisfies the given condition, if any elements do satisfy it. If none do, then @ still chooses some arbitrary element. This is a total function, so it always returns the same choice, but all that is known about the element chosen is the property specified, and that only if there exists such an element.

Given this definition, we can prove that it is well-defined, in the sense that every pair of the sequence satisfies the definition property, as in Table 11.7.

$$
\begin{array}{l}
\vdash \forall i\ p_1\ s_1\ \rho. \\
\qquad WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ p_1))\ s_1\ \wedge \\
\qquad \sim(terminates\ p_1\ s_1\ \rho)\ \Rightarrow \\
\qquad \textbf{let}\ (p_2, s_2) = mk\_sequence\ i\ \rho\ p_1\ s_1\ \textbf{in} \\
\qquad (Depth\_calls\ i\ p_1\ s_1\ p_2\ s_2\ \rho\ \wedge\ \sim(terminates\ p_2\ s_2\ \rho))
\end{array}
$$

Table 11.7: Definitional property satisfied by $mk\_sequence$.

The most important property we prove about *mk_sequence* is that the sequence it generates is chained together by each consecutive pair being related by one level of procedure call, as expressed in Table 11.8.

$$
\begin{array}{l}
\vdash \forall i\ p_1\ s_1\ \rho. \\
\qquad WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ p_1))\ s_1\ \wedge \\
\qquad \sim(terminates\ p_1\ s_1\ \rho)\ \Rightarrow \\
\qquad\ \ \mathbf{let}\ (p_2, s_2) = mk\_sequence\ i\ \rho\ p_1\ s_1\ \mathbf{in} \\
\qquad\ \ \mathbf{let}\ (p_3, s_3) = mk\_sequence\ (i+1)\ \rho\ p_1\ s_1\ \mathbf{in} \\
\qquad\ \ M\_calls\ p_2\ s_2\ [\,]\ p_3\ s_3\ \rho
\end{array}
$$

Table 11.8: Chain of calls induced by *mk_sequence*.

Given this generator function *mk_sequence*, it is possible to prove that the sequence of procedure names and states it generates satisfies the properties in Table 11.9 to be called an infinite recursive descent sequence.

$$
\begin{array}{l}
sequence\ ps\ sts\ ns\ \rho\ \ = \\
\qquad (\forall i.\ M\_calls\ (ps\ i)\ (sts\ i)\ [\,]\ (ps\ (i+1))\ (sts\ (i+1))\ \rho)\ \wedge \\
\qquad (\forall i.\ ns\ i\ \ =\ \ (\mathbf{let}\ \langle vars, vals, glbs, pre, post, calls, rec, c\rangle = \rho\ (ps\ i)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad induct\_start\_num\ (sts\ i)\ rec))
\end{array}
$$

Table 11.9: Infinite Recursive Descent Sequence Predicate *sequence*.

In this definition, *ps* is an infinite sequence of procedure names, represented as a function from `num` to `string`. The number used as the index is the depth number from *Depth_calls*. *ps* contains the infinite sequence of names of procedures called in the hypothesized infinite recursive descent; it is the path downward.

Likewise, sts is the corresponding infinite sequence of states, each one the state reached in the corresponding procedure in ps in the process of the infinite recursive descent.

Finally, ns is the corresponding infinite sequence of the values of the recursion expressions of each procedure in ps, evaluated in the corresponding state given in sts. Several procedures may be represented in this list; we shall see that for the subsequences of this sequence for any particular procedure, each such subsequence will be strictly decreasing. The values in ns are generated by the function $induct\_start\_num$, defined in Table 11.10.

$$induct\_start\_num\ s\ \textbf{false} \;=\; 0$$

$$induct\_start\_num\ s\ (v < x) \;=\; V\ v\ s$$

Table 11.10: Recursion Expression Value Function $induct\_start\_num$.

This gives the definition of an infinite recursive descent sequence. Such a sequence is implied by the assumption stated earlier, that there does not exist any n such that all calls of depth n or less terminate. We can now prove this as the theorem listed in Table 11.11, using $mk\_sequence$ to create an explicit witness.

$$
\begin{array}{l}
\vdash \forall p_1\ s_1\ \rho. \\
\qquad WF_{env\_term}\ \rho \wedge WF_{env\_pre}\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ p_1))\ s_1\ \wedge \\
\qquad \sim(\exists n.\ \forall m\ p_2\ s_2.\ m \leq n\ \ \wedge\ \ Depth\_calls\ m\ p_1\ s_1\ p_2\ s_2\ \rho \Rightarrow \\
\qquad\qquad\qquad terminates\ p_2\ s_2\ \rho)\ \Rightarrow \\
\qquad (\exists ps\ sts\ ns.\ sequence\ ps\ sts\ ns\ \rho\ \ \wedge\ \ (ps\ 0 = p_1)\ \ \wedge\ \ (sts\ 0 = s_1))
\end{array}
$$

Table 11.11: Existence of Infinite Recursive Descent Sequence.

11.2.4 Consequences of an Infinite Sequence

One consequence of the definition of an infinite recursive descent sequence is that any two points in the sequence are related by *M_calls*; this is displayed in Table 11.12.

$$
\begin{array}{l}
\vdash \forall \rho\ ps\ sts\ ns\ i\ j. \\
\qquad sequence\ ps\ sts\ ns\ \rho\ \ \wedge\ \ i < j\ \ \Rightarrow \\
\qquad (\exists ps'.\ M\_calls\ (ps\ i)\ (sts\ i)\ ps'\ (ps\ j)\ (sts\ j)\ \rho)
\end{array}
$$

Table 11.12: Sequence calls related by *M_calls*.

Preconditions are maintained across points in the sequence, as in Table 11.13.

$$
\begin{array}{l}
\vdash \forall i\ \rho\ ps\ sts\ ns. \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \ \Rightarrow \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ i)))\ (sts\ i)
\end{array}
$$

Table 11.13: Sequence Precondition Maintenance.

11.2.5 Strictly Decreasing Sequences

Perhaps the most important consequence of an infinite recursive descent sequence results from combining it with the knowledge contained in the recursiveness property, WF_{env_rec}. WF_{env_rec} says that every recursive call of a procedure exhibits the strict decrease of the value of its recursion expression. For sequences, this gives us the ability to prove the theorem in Table 11.14. This says that for any two points in the infinite sequence which refer to the *same* procedure, the value of the recursion expression as stored in ns strictly decreases.

$$
\begin{array}{l}
\vdash \forall \rho\ ps\ sts\ ns\ p\ i\ j\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\qquad WF_{env\_syntax}\ \rho\ \wedge \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad WF_{env\_rec}\ \rho\ \wedge \\
\qquad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \wedge \\
\qquad (\rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle)\ \wedge \\
\qquad (ps\ i = p)\ \wedge \\
\qquad (ps\ j = p)\ \wedge \\
\qquad i < j\ \Rightarrow \\
\qquad (ns\ j < ns\ i)
\end{array}
$$

Table 11.14: Sequence Decreasing Values.

To make use of this strictly decreasing property, we choose a minor variation on the proof sketch described earlier. Instead of claiming that there must be some procedure which has an infinite number of occurrences in the sequence, we take the approach of proving that every procedure has only a finite number of occurrences in the sequence. We first prove that given any occurrence of a procedure p in the sequence, there is a maximum limit on the index of the elements beyond which none of the elements refer to that same procedure p, as shown in Table 11.15.

$$
\begin{array}{l}
\vdash \forall n\ i\ p\ \rho\ ps\ sts\ ns\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\qquad WF_{env\_syntax}\ \rho\ \wedge \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad WF_{env\_rec}\ \rho\ \wedge \\
\qquad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \wedge \\
\qquad (\rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle)\ \wedge \\
\qquad (ps\ i = p)\ \wedge \\
\qquad (ns\ i = n)\ \Rightarrow \\
\qquad (\exists m.\ \forall j.\ m < j \Rightarrow ps\ j \neq p)
\end{array}
$$

Table 11.15: Sequence Occurrence Implies Limit on Occurrences.

This is proven by well-founded induction on n, the value of the recursion expression at the ith procedure in the sequence, making use of the fact that the values of the recursion expression are members of a well-founded set.

From this we are able to prove that for every procedure, there is a maximum limit on the index of the elements which refer to it, as shown in Table 11.16.

$$
\begin{array}{l}
\vdash \forall p\ \rho\ ps\ sts\ ns. \\
\qquad WF_{env\_syntax}\ \rho\ \wedge \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad WF_{env\_rec}\ \rho\ \wedge \\
\qquad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \Rightarrow \\
\qquad (\exists m.\ \forall j.\ m < j \Rightarrow ps\ j \neq p)
\end{array}
$$

Table 11.16: Each Procedure Has Limit on Occurrences.

Next we need to establish that every procedure in the sequence is a member of the finite list of defined procedures, all_ps, as described in Table 11.17.

$$
\begin{array}{l}
\vdash \forall all\_ps\ \rho\ ps\ sts\ ns. \\
\quad WF_{env\_pre}\ \rho\ \wedge \\
\quad (\forall p.\ p \notin SL\ all\_ps \Rightarrow \rho\ p = \rho_0\ p)\ \wedge \\
\quad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\quad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \Rightarrow \\
\quad (\forall i.\ ps\ i \in SL\ all\_ps)
\end{array}
$$

Table 11.17: Each Procedure in Sequence is in all_ps.

We can now prove that since for each procedure there is a maximum limit on its occurrences in ps, and since there is only a finite number of procedures, there must be a maxiumum limit on the sequence as a whole. This means there exists a single limit m which bounds the indices of the occurrences of *all* the procedures listed in all_ps, as in Table 11.18.

$$
\begin{array}{l}
\vdash \forall all\_ps\ \rho\ ps\ sts\ ns. \\
\quad WF_{env\_syntax}\ \rho\ \wedge \\
\quad WF_{env\_pre}\ \rho\ \wedge \\
\quad WF_{env\_rec}\ \rho\ \wedge \\
\quad sequence\ ps\ sts\ ns\ \rho\ \wedge \\
\quad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \Rightarrow \\
\quad (\exists m.\ \forall p.\ p \in SL\ all\_ps \Rightarrow (\forall j.\ m < j \Rightarrow ps\ j \neq p))
\end{array}
$$

Table 11.18: Limit on All Occurrences in all_ps.

This then contradicts the assumption of the infinite sequence, since there are many elements beyond the maximum limit, and they must belong to *some* defined procedure. This contradiction is expressed in Table 11.19.

Given this contradiction, implied by the assumption that there did not exist

$$
\begin{array}{l}
\vdash \forall \rho\ all\_ps\ ps\ sts\ ns. \\
\qquad WF_{env\_syntax}\ \rho\ \wedge \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad WF_{env\_rec}\ \rho\ \wedge \\
\qquad (\forall p.\ p \notin SL\ all\_ps \Rightarrow \rho\ p = \rho_0\ p)\ \wedge \\
\qquad A\ ((FST \circ SND \circ SND \circ SND)\ (\rho\ (ps\ 0)))\ (sts\ 0)\ \Rightarrow \\
\qquad \sim(sequence\ ps\ sts\ ns\ \rho)
\end{array}
$$

Table 11.19: Sequence Contradiction.

any n such that all calls of depth n or less terminated, we can conclude that such an n must exist, and hence by the theorem in Table 11.4, we can prove that every procedure terminates, as shown in Table 11.20.

$$
\begin{array}{l}
\vdash \forall \rho\ all\_ps\ p\ s\ vars\ vals\ glbs\ pre\ post\ calls\ rec\ c. \\
\qquad WF_{env\_syntax}\ \rho\ \wedge \\
\qquad WF_{env\_pre}\ \rho\ \wedge \\
\qquad WF_{env\_rec}\ \rho\ \wedge \\
\qquad WF_{env\_term}\ \rho\ \wedge \\
\qquad (\forall p.\ p \notin SL\ all\_ps \Rightarrow \rho\ p = \rho_0\ p)\ \wedge \\
\qquad (\rho\ p = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle)\ \wedge \\
\qquad A\ pre\ s\ \Rightarrow \\
\qquad terminates\ p\ s\ \rho
\end{array}
$$

Table 11.20: Procedure Termination.

Finally, given the termination of each procedure when called, we can prove the total correctness of the entire environment of procedures, as in Table 11.21.

$$
\begin{array}{|l|}
\hline
\\
\forall d\ \rho. \\
\quad \rho = mkenv\ d\ \rho_0\ \wedge \\
\quad WF_{env\_syntax}\ \rho\ \wedge \\
\quad WF_{env\_pre}\ \rho\ \wedge \\
\quad WF_{env\_rec}\ \rho\ \wedge \\
\quad WF_{env\_term}\ \rho\ \Rightarrow \\
\quad WF_{env\_total}\ \rho \\
\\
\hline
\end{array}
$$

Table 11.21: Total Correctness of Procedure Environment.

This completes our proof of termination for the Sunrise language.

Part IV

Conclusions

CHAPTER 12

Significance

"LORD, make me to know my end,

And what is the measure of my days,

That I may know how frail I am."

— Psalm 39:4

In this chapter we will reflect and explore the significance of this work, and the possibility of its usefulness in the future.

The most novel part of this work is the development of a new methodology for proving the termination of programs with mutually recursive procedures. This includes new specifications to include in the headers of procedures, an algorithm for analyzing the procedure call graph to produce verification conditions, and logics for proving the termination of procedures from those verification conditions. We feel the approach is easier and simpler to use than previous proposals, while being more general in the sense of providing natural proofs of termination related to the program's original purpose. It also regularizes the proofs, making each example's proof less *ad hoc*, and structuring the proof according to the program logics. Furthermore, this methodology can be automated by a VCG, as we have done and exhibited in Chapter 8. This methodology should in general translate to other programming languages, and we see this as a valuable technology for proving the termination of programs with procedures.

The most central thing we have learned from this work has been that the general approach we used was feasible. It was powerful, in that we could prove meta-theorems about all Sunrise programs, and it was effective, in that those proofs were accomplished once and would not need to be repeated for each application of the VCG. It was also quite difficult, in that there was considerable effort and skill required to accomplish the verification of the VCG.

In addition, the approach is quite solidly sound. Everything was established from the ground up, without claiming any new axioms, and only extending the theory by new definitions. Because we constructed a deep embedding of the programming and assertion languages within HOL, the types used to represent the abstract syntax trees were new types, without connections to or dependencies on previous parts of the theory. We established the semantics of the syntax trees ourselves by defining the operational semantics of the programming language and the denotational semantics of the assertion language. These semantics are simple and easily examined by the community, with their implications more easily understood than if we had taken an axiomatic semantics as the foundational definition. Then the axioms and rules of the axiomatic semantics were proven as theorems from the underlying foundational semantics, ensuring their soundness. Based on these sound axioms and rules, the VCG functions were verified and proven to be sound, which is our primary result.

The definitions and proofs are even more solidly secured by having created them within the HOL theorem proving environment, which ensures the soundness of any theorems proven using its tools. For a user who is able to find the path to the goal of proving a theorem, HOL presents a powerful confirmation that that proof is in fact valid. HOL is generally understood to be weak in automating the search for a proof, say as compared with the Boyer-Moore theorem prover. Nevertheless, it was powerful and effective enough for our purposes here. Therefore, we can claim with assurance that this proof of soundness of this VCG has no logical errors. We have great confidence that the HOL-implemented proof is completely sound and trustworthy, and by extension, that the proofs of any programs proved using the VCG are likewise completely sound and trustworthy.

The idea of using a verification condition generator seems a useful and practical one, but this idea will need to be verified by actual experimentation and experience. The VCG we defined for total correctness seems quite satisfactory when it comes to the traditional analysis of the syntax of the program; there is room for improvement in the analysis of the call graph structure, as is discussed in Chapter 14.

The programming and assertion languages considered were quite small and not suitable for actual programming. This is because our goal was the exploration of the ideas behind certain program constructs, principally recursive procedures, and we included features that supported that goal. Nevertheless, it is not difficult to see how the languages could be extended with a more complete assortment of operators. This will be explored more in the next chapter.

The handling of expressions with side effects by the use of translation functions was elegant and surprisingly easy, once we had decided to use simultaneous substitutions to represent changes to the state. This part of the work has been quite successful in handling our simple expressions. Future work will explore the applicability of this approach to more complex side effects.

The entrance and termination logics arose naturally during our work, and became the most convenient way to establish the verification of programs and the VCG itself. These are restricted versions of temporal logic, but powerful enough to accomplish the proofs of the recursiveness properties and the termination of procedures. It was important for us to develop some constraints on temporal logic, else it would not have been feasible to write a simple VCG to prove hypotheses written in such an expressive language.

Several realizations arose during the course of this work, and we present them here as understandings we have developed. These concern the separation of the programming and assertion languages, the need for well-formedness predicates, and the significant gap between partial and total correctness.

We believe that it is important to keep the ideas of the programming and assertion languages separate, and not confuse them, even if one's language does not include expressions with side effects. These two languages have different qualities and purposes, as was explored at the end of Section 5.5. One should not be beguiled by their overlap in appearance into assumming they are the same in essence.

Despite the relative lack of attention paid to date to well-formedness, we found this to be an area requiring a significant portion of the total effort. Perhaps the goal of complete formal verification of this system in every detail forced us to look at issues that previously were easy to dismiss. Just because an issue is obvious and part of common sense, does not mean that its formal verification is inconsequential, either in effort required or in significance of the results. It appears to us that well-formedness will need to be a part of any practical VCG constructed in the future.

Finally, we feel that this work explores in a thorough way the difference between partial and total correctness of programs with mutually recursive procedures. The specifications required of the user for each procedure differed for specifying their partial correctness claims, using "**pre**" and "**post**", and their termination claims, using "**calls** ... **with**" and "**recurses with**". A respectable fraction of the total structure of the proof was principally concerned with proving total correctness; three out of the five program logics used were principally devoted to proving either termination or total correctness. Also, the structure of the proofs of partial and total correctness differed markedly. The proof of partial correctness worked by stages, proceeding by normal mathematical induction on the depth of recursive call to prove the entire environment well-formed for partial correctness. In contrast, the proof of total correctness involved an exploration of the procedure call graph to identify procedure call cycles and produce verification conditions which established the progress achieved around each cycle. Termination then followed based on a well-foundedness argument about infinitely decreasing sequences.

Clearly our tool would not be suitable for proving programs correct in an industrial setting. Rather, this has been a theoretical exploration of ideas in building a solid foundation for program proofs. In the future, these ideas may be of use to other researchers in building practical verification condition generators to help prove real programs.

CHAPTER 13

Ease of Use

"For My yoke is easy and My burden is light."

— Matthew 11:30

In this chapter we consider the ease of use of the Sunrise system for proving programs correct. This includes the burdens of the annotations required for while loops and procedures, and the burdens of proving the verification conditions created. We also discuss the areas of the proof that the VCG supports.

13.1 Burden of Annotation

To prepare a program for submission to the VCG, the Sunrise system requires the user to attach a number of annotations to the program which have no direct impact on the program's execution, and serve only to help the VCG and the proof of the program's correctness. It is reasonable to ask how burdensome these required annotations are, how much is asked of the user, and how a user might be expected to generate such annotations in practice.

Most of these questions are similar to the ones raised in the debate over loop invariants, whether or not the user should be expected to contribute the loop invariants, and the apparent difficulty of such a task. It has been argued that requiring the user to provide such invariants forces the user to think more clearly about why they should be true, and that they also provide a very useful form of documentation. We consider the question of the propriety of requiring invariants, and other annotations, to be a decision beyond the purview of this work. In this work, requiring invariants and other annotations is a pragmatic necessity. We now examine the difficulty of arriving at such annotations, considering each one in turn.

For while loops, two annotations are generally required, a loop invariant and a loop progress expression containing an expression whose value strictly decreases for each iteration of the loop. The invariant is used to prove the partial correctness of the loop, and the progress expression is used to prove its termination. Gries has studied the problem of generating loop invariants [Gri81] and arrived at a number of principles to guide this task. He has also described how to generate a progress expression (which he calls a bound function) so that each iteration makes progress towards termination.

For procedure declarations, we require several annotations:

1. Global variables

2. Precondition

3. Postcondition

4. For each procedure called in the body, a *calls* progress expression

5. If the procedure recurses, a recursion progress expression.

The burden of generating a complete list of global variables is not hard, but it is not as simple as scanning the body of the procedure. Instead, this should include all globals accessed from within procedures called from within the body of this procedure, either directly or indirectly, any number of levels deep. Thus, the globals list should be a list of all globals that can be read or written during the execution of the procedure body. If procedures are written in a bottom-up fashion, then this would be the union of the globals lists of all procedures called by the body, together with the globals actually used in the body itself.

The specifications of the precondition and postcondition are well-discussed in the literature, and will not be described further here.

The new specification of the *calls* progress expressions expresses a connection between two states, in some ways analogous to the connection expressed by post-conditions. Here, however, we need to take care to refer to the correct variables in the two contexts. The choice of these progress expressions is crucial to the proof of termination, for these are used to generate the path conditions while traversing the procedure call graph, and in creating the call graph verification conditions. These may be created by asking the question, "What sort of progress do I expect to achieve between the entrance of this procedure and the entrance of another called by this one?" We suggest first drawing the procedure call graph and examining it for cycles, to manually focus one's attention on the need to provide meaningful progress towards termination around each cycle. This progress is then expressed in the recursion expression of the procedure. The progress around each cycle then needs to be broken down into smaller steps of progress, which are distributed onto the various arcs of the graph. These smaller steps may in fact individually show no progress, or even backwards movement as long as it is limited, as may be convenient. The requirement is that the accumulation of the progress of all the arcs around a cycle must show the forward progress of the recursive progress expression. Thus the choice of the recursive progress expression should precede the choice of the *calls* progress expressions.

The need to specify these calls progress expressions and the recursion expression in each procedure's header is welcome, for it compels the programmer to think seriously about the issues of termination for his program. For every possible path of recursion, there must be progress towards termination that can be identified and quantified. Usually this progress will be nascent within the programmer, as part of his design of the program, but the annotation requirements will force him to make these ideas concrete, and to examine them critically. In cases of great interaction among procedures, where the procedure call graph has many interlocking cycles, the expectation of having to prove termination may draw the programmer toward simplified designs with fewer well-chosen interactions.

This annotation structure was chosen as a compromise between the simple rigidity of Sokolowski's recursion depth counter, and the extreme flexibility of specifying the expected progress individually for each call, at the point of call. We chose to require every call issuing from one particular procedure to another to satisfy the same progress condition. This allowed us to partition the proof of recursion into two stages, where in the first stage the calls progress claims were verified by syntactic analysis of each procedure's body, and in the second stage, the recursion progress claims were verified from the calls progress claims by analyzing the structure of the call graph. This followed the compositional paradigm, where the proof of each individual procedure was accomplished in relative isolation, and then the results of these proofs were brought together to verify the entire collection of procedures.

We feel this is a reasonable annotation structure, because if the programmer wished to prove termination, inherently he would have to describe how to prevent infinite recursive descent, and this leads immediately to a consideration of cycles in the procedure call graph. Each such cycle must be shown to terminate, probably by some form of a well-founded argument. Inevitably the programmer would have to supply information similar to what we have asked for in these annotations, and not having considered the issue beforehand, might choose a simple but overly restrictive system like recursion depth counters. Requiring our annotations at the beginning brings the programmer's attention to termination issues early, and clarifies the expectations of progress between procedures. Therefore this annotation structure would be a welcome element in good software engineering and modular design for implementation by a team.

13.2 Burden of Proof

The verification conditions presented by the part of the VCG that deals with analyzing the syntactic structure of the program appears to be quite satisfactory. However, the production of verification conditions sufficient to establish termination, created by analyzing the structure of the call graph, may allow for substantial reduction in the number of verification conditions generated. One such improvement is discussed in Chapter 14. This may be the subject of a future upgrade of Sunrise.

13.3 Areas of VCG Support

To briefly mention the concepts proven automatically by the VCG without user involvement, the user need not be concerned with proving

1. well-formedness
2. proof by stages of partial correctness
3. precondition maintenance
4. *calls* progress
5. recursive progress
6. termination
7. total correctness

All of these follow from simply proving the verification conditions. We do not mean to imply that the proof of the verification conditions is trivial or easy. They may well contain the bulk of the weight of the proof. However, the above concepts are not themselves trivial, and we contend that this VCG as presented does accomplish a significant task in reducing the difficulty of proving programs totally correct.

CHAPTER 14

Future Research

"Thus says the LORD,

The Holy One of Israel, and his Maker:

'Ask Me of things to come concerning My sons;

And concerning the work of My hands, you command Me.' "

— Isaiah 45:11

"Whatever He hears He will speak; and He will show you things to come."

— John 16:13

In this chapter we consider possible future developments of the ideas presented in this work. These fall into four major areas: extensions to the programming and assertion languages, improvements to the VCG, implementations and tools to support the methodologies presented here, and proofs of completeness.

14.1 Language Extensions

There are many areas where we would like to extend the programming and assertion languages described here.

Probably the most immediate need is the inclusion of arrays. It is difficult to arrive a general, useful examples without arrays. This topic has been studied extensively before, so it should pose few theoretical difficulties. Some of the issues involved concern the inclusion of array bound checks in the preconditions computed by the VCG, the extension of the concept of aliasing to forbid confusion between array elements, and the passing of entire arrays as parameters.

The progress expressions currently permitted allow only the use of the operator $<$, implying the well-founded set of nonnegative integers. We expect to extend this to include the operator $\ll$, with the well-founded set of lists of nonnegative integers ordered lexicographically, and to include other well-founded sets and ordering relations. There does not appear to be any fundamental difficulty in adapting the proofs of recursiveness or termination to these additional forms. They would provide the ability to prove the termination of a wider variety of programs in ways that are natural and appropriate to the subjects of the programs.

In order to prove programs that implement certain recursive functions such as Ackerman's function, it will be necessary to extend the assertion language with user-defined functions, defined solely within the assertion language in order to abstract parts of the specifications. Even if no recursive functions are needed, such user-defined functions will be very practically useful in clearly expressing complex and layered specifications.

Many new operators can be added in a similar style to those already present. For example, if we add operators to perform integer division and check whether a number is odd or even, we can run Pandya and Joseph's example. In general, this seems to be one of the simplest and easiest extensions to accomplish, needing no theoretical additions. Nevertheless, we have not at this time expanded the language unnecessarily because of the great time and space issues that arise when defining new types in HOL which have many cases to represent the syntax trees.

One area of particular interest is the area of typing. A first extension would focus on adding valuable new base types, such as characters, strings, or bounded integers, for which there already exists support in the HOL logic. Further extensions could explore the creation of structured types such as records and arrays.

Input and output are important in bringing these systems closer to reality. We can model these as undetermined assignments to particular global variables, with assertions to act as preconditions restricting the possible input sequences. We would like to explore if the same translation techniques now used for the increment operator will also support input as an undetermined assignment.

One of the greatest challenges facing program verification is scaling up the theory to handle large, or even medium-sized programs, say of several tens of thousands of lines long. Possibly the only means will be through a form of modularization, where some program construct like Ada packages or Modula-2 modules will be used to encapsulate a section of the program with a well-defined interface. In the past these interfaces have incorporated only a syntactic specification, of the arity of each procedure and the types of its parameters. In the future we envision interfaces specifying the behavior and meaning of each module, just as preconditions and postconditions express that for procedures in this work. The point of the encapsulation is to modularize the proof of correctness of the program as well. Following the structure of the program, the proof should be structured so that each module can be independently verified apart from the rest of the program, perhaps with some required context as a precondition. Then the proofs of the verified modules should be adaptable for completing proofs of other parts of the program that use the modules. This situation is analogous on a larger scale to the specification and use of procedures in this work.

One of the most intriguing aspects of programming languages is nondeterminism, where either the order of subexpressions or the value of the operator itself may vary from one execution to the next. We would like to introduce an operator which nondeterministically selects an integer from 1 to n, so as to explore nondeterminism from the level of expressions up. Dijkstra's guarded conditional and repetition commands would be included as well. Nondeterminism may be handled by the same type of predicates for the operational semantics as are currently used; the final state will simply no longer be uniquely determined, but in fact these predicates will become true relations.

Finally, we hope someday to investigate the theoretically difficult area of concurrency. Concurrency raises a host of new issues, ranging from the level of structural operational semantics ("big-step" versus "small-step"), to dealing with assertions describing temporal sequences of states instead of single states, to issues of fairness. We believe that a proper treatment of concurrency will exhibit qualities of modularity and compositionality. *Modularity* means that a specification for a process should state both (a) the assumptions under which it should operate, and (b) the task (or commitment) which it should meet, given those assumptions. *Compositionality* means that the specification of a system of processes should be verifiable in terms of the specifications of the individual constituent processes.

14.2 VCG Improvements

We intend to continue to examine and improve the VCG functions for greater efficiency and ease of use, for example to reduce the number of verification conditions generated, especially those created through the analysis of the procedure call graph. One immediate improvement may be found by generating the verification conditions for each procedure in order. When the termination of a procedure was thus established, it would be deleted from the procedure call graph along with all incident arcs. This smaller call graph would then be the one used in generating verification conditions for the next procedure in order. Since there would be fewer arcs, there would be fewer cycles, and we anticipate far fewer verification conditions produced.

14.3 Implementations

We envision the theory developed in this work and others being supported by a variety of tools to ease the process of creating verified software. Proving programs correct is sufficiently difficult and full of details that mechanizing the task is a natural goal.

One tool would be a program editor, which would act as a structured editor for creating programs, but when a sufficiently substantial part was created (for example, a procedure) it would then automatically invoke the VCG on it. Then the verification conditions it produced would be collected and presented to the user to solve. The system could enforce the constraint that until all verification conditions were proven by the user, the code would not be submitted to the compiler, and thus could not be run.

In order to aid the user in proving these verification conditions, substantial theorem proving systems will have to be presented. We anticipate powerful graphical user interfaces to pictorially diagram the user's search for the correct proof. These would complement semi-automatic theorem provers running in the background, which would search for proofs of simple verification conditions or simple subgoals of a larger proof. This would eliminate the lower branches of the proof tree from the user's attention; and for most trees the lower branches contain the bulk of the tree's structure.

14.4 Completeness

Although we have not attempted any proof of completeness of this proof system, that does not mean that we think that unimportant. In the future we hope to create a proof of the system's *relative completeness*, in the sense of Cook [Coo78]. To some degree this will induce modifications of this approach, for completeness is a statement of what can be proven about a true program, and this would require encapsulating a proof system inside HOL.

CHAPTER 15

Conclusions

"You shall know* the truth, and the truth shall make you free."
— John 8:32

"But now having been set free from sin, and having become slaves of God, you have your fruit to holiness, and the end, everlasting life."
— Romans 6:22

* "**know,** *ginosko* (ghin-*oce*-koe); Strong's #1097: To perceive, understand, recognize, gain knowledge, realize, come to know. *Ginosko* is the knowledge that has an inception, a progress, and an attainment. It is the recognition of truth by personal experience."
— The Spirit-Filled Life Bible, Thomas Nelson Publishers, 1991, page 1589.

We have presented in this dissertation a verification condition generator tool for proving programs totally correct. We have verified the VCG, proving it sound from a foundation of a structural operational semantics. From this operational semantics we derived an axiomatic semantics, as theorems whose soundness was established by proof. From these we proved the correctness of the VCG. The entire proof has been conducted within the HOL mechanical theorem proving environment, guaranteeing the soundness of the reasoning and the verification result.

As part of this process, we developed five program logics, three of which were fundamental new inventions in this work, namely the expression logic, the entrance logic, and the termination logic. These regularized the process of proving termination for a program with mutually recursive procedures, and formed a structure less *ad hoc* than previous proposals.

This work has now provided a tool which can substantially decrease the difficulty of proving programs correct. It does not eliminate that difficulty, and even the use of this tool requires training and expertise. However, it points the direction towards mechanical assistance of the proof process which we believe is essential to the practical realization of the dream of widespread program verification. Such tools must not only be powerful and efficient, but it is vital that they themselves be trustworthy, for the proofs constructed using those tools can be no more reliable than the tools themselves.

This trustworthiness is now demonstrated to be feasible, by the presentation of this VCG tool. We believe that the annotation structure described is not onerous, but reasonable and intuitive. It is extremely important that whatever structure is imposed aids, and does not obstruct, the creation process. We have attempted to craft the annotation structures described in this work to be simple and structurally well placed, so as to provide the maximum strength with the minimum constraint. Extending this work to new language features and styles will require new annotation and proof structures. We look forward to further developments for greater strength in days to come.

References

[AA78] Suad Alagić and Michael A. Arbib. *The Design of Well-Structured and Correct Programs.* Springer-Verlag, 1978.

[AdB90] Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84(2):129–162, February 1990.

[Age91] Sten Agerholm. Mechanizing program verification in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 208–222. IEEE Computer Society Press, August 1991.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, New York, 1991.

[Apt81] K. R. Apt. Ten years of hoare logic: A survey—part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[BGG+92] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and T. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. Elsevier Science Publishers B.V. (North Holland), 1992.

[BM81] Robert S. Boyer and J Strother Moore. A verification condition generator for FORTRAN. In Robert S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science.* Academic Press, London, 1981.

[BM88] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[CM92] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

[Dah92] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, London, 1992.

[dB80] Jaco de Bakker. *Mathematical Theory of Program Correctness*. Prentice Hall International, London, 1980.

[Dij72] Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of the American Mathematical Society Symposia in Applied Mathematics*, volume 19, pages 19–31, Providence, R.I., 1967. American Mathematical Society.

[Fra92] Nissim Francez. *Program Verification*. Addison-Wesley, Wokingham, England, 1992.

[GM93] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[Gor88] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall International Series in Computer Science. Prentice Hall, London, 1988.

[Gor89] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. In P. A. Subrahmanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–489. Springer-Verlag, New York, 1989.

[Gra87] D. Gray. A pedagogical verification condition generator. *The Computer Journal*, 30(3):239–248, June 1987.

[Gri81] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.

[HM94] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In Thomas F. Melham and Juanito Camilleri, editors, *Proceedings of the 1994 International Workshop on the HOL Theorem Proving System and its Applications*, pages 269–284. Springer-Verlag, September 1994. LNCS 859.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–581, October 1969.

[Hoa71] C. A. R. Hoare. Procedures and parameters: an axiomatic approach. In E. Engeler, editor, *Proceedings of Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, Berlin, 1971. Springer-Verlag.

[ILL75] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification: A logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.

[Kau94] Matt Kaufmann. Combining an interpreter-based approach to software verification with verification condition generation. Technical Report 97, Computational Logic, Inc., April 1994.

[Lin93] H. Lin. A verification tool for value-passing processes. In *Proceedings of PSTV XIII, Liege, Belgium*, May 1993.

[Mel89] Thomas F. Melham. Automating recursive type definitions in higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[Mel91] Thomas F. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL*

Theorem Proving System and its Applications, pages 350–357. IEEE Computer Society Press, August 1991.

[Mel92] Thomas F. Melham. A mechanized theory of the π-calculus in HOL. Technical Report 244, University of Cambridge Computer Laboratory, January 1992.

[Nes93] Monica Nesi. Value-passing CCS in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Proceedings of the HUG'93 6th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pages 352–365. Springer-Verlag, August 1993. LNCS 780.

[PJ86] P. Pandya and M. Joseph. A structure-directed total correctness proof rule for recursive procedure calls. *The Computer Journal*, 29(6):531–537, 1986.

[Plo81] Gordon Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University Computer Science Department, September 1981.

[Rag73] Larry Calvin Ragland. A verified program verifier. Technical Report 18, Department of Computer Sciences, The University of Texas at Austin, Austin, May 1973.

[Sok77] Stefan Sokołowski. Total correctness for procedures. In J. Gruska, editor, *Proceedings, 6th Symposium on the Mathematical Foundations of Computer Science*, pages 475–483. Springer-Verlag, September 1977. LNCS 53.

[Sok84] Stefan Sokołowski. Partial correctness: The term-wise approach. *Science of Computer Programming*, 4:141–157, 1984.

[Sto88] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[ZSO+93] Cui Zhang, Rob Shaw, Ronald A. Olsson, Karl Levitt, Myla Archer, Mark R. Heckman, and Gregory D. Benson. Mechanizing a programming logic for the concurrent programming language microSR in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving* *and its Applications*, number 780 in LNCS, pages 29–42. Springer-Verlag, August 1993.