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

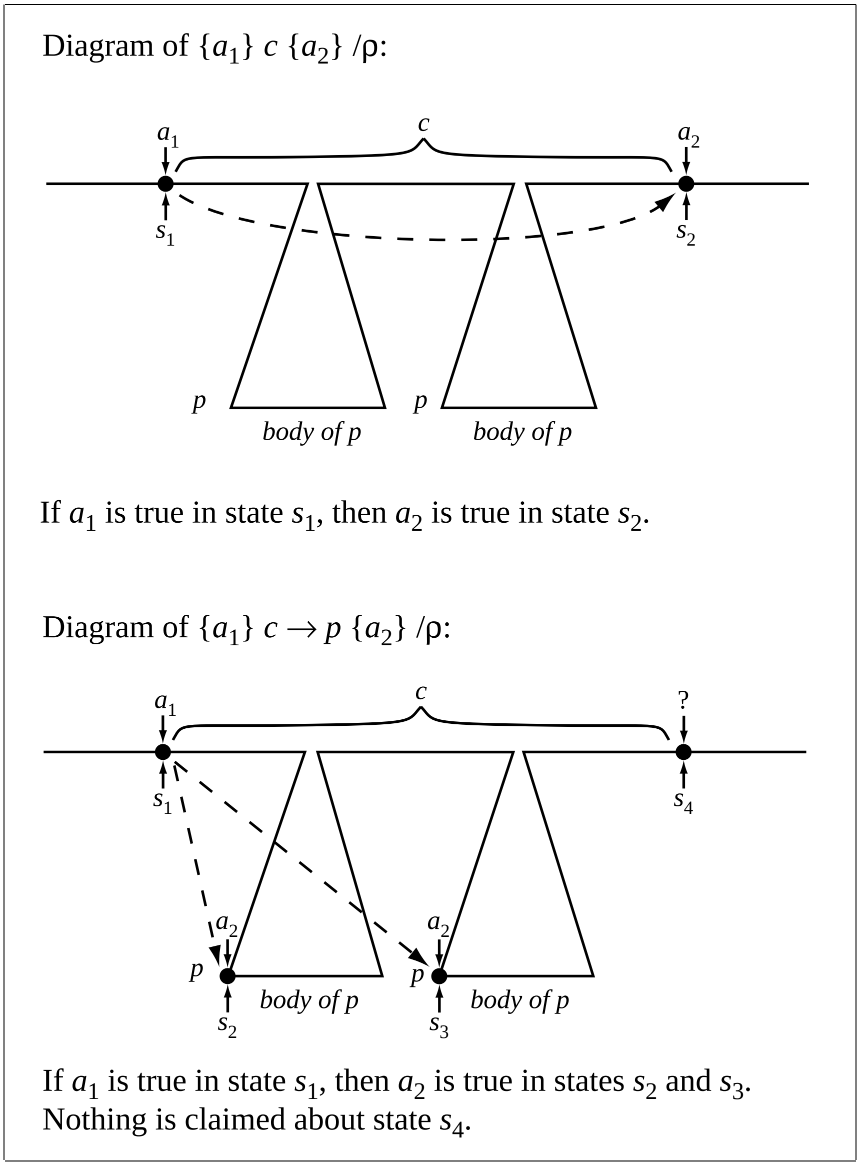

Diagram of $\{a_1\}\ c\ \{a_2\}\ /\rho$:

If a_1 is true in state s_1, then a_2 is true in state s_2.

Diagram of $\{a_1\}\ c \rightarrow p\ \{a_2\}\ /\rho$:

If a_1 is true in state s_1, then a_2 is true in states s_2 and s_3.
Nothing is claimed about state s_4.

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 $\{a_1\}\ c \to p\ \{a_2\}\ /\rho$, 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    $(\exists b.\ \widehat{n} = 2 * b + a) \ \wedge \ a < 2 \ \wedge \ n = \widehat{n}$;
        calls   $odd$     with $n < \widehat{n}$;
        calls   $even$    with $n < \widehat{n}$;
        recurses          with $n < \widehat{n}$;

        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    $(\exists b.\ \widehat{n} + 1 = 2 * b + a) \ \wedge \ a < 2 \ \wedge \ n = \widehat{n}$;
        calls   $even$    with $n < \widehat{n}$;
        calls   $odd$     with $n < \widehat{n}$;
        recurses          with $n < \widehat{n}$;

        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
[ $a = 1$ ]
```

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:

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

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

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

False Precondition:

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

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

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

Postcondition Weakening:

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

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

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

Numeric Expression Precondition:

$$\frac{}{[\mathit{ae\_pre}\ e\ q]\ e\ [q]}$$

Expression List Precondition:

$$\frac{}{[\mathit{aes\_pre}\ es\ q]\ es\ [q]}$$

Boolean Expression Precondition:

$$\frac{}{[\mathit{ab\_pre}\ b\ q]\ b\ [q]}$$

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:

$$\frac{}{[q]\ n\ [q]}$$

Variable:

$$\frac{}{[q]\ x\ [q]}$$

Increment:

$$\frac{}{[q \lhd [(x+1)/x]]\ \ {+}{+}\,x\ [q]}$$

Addition:

$$\frac{\begin{array}{c}[p]\ e_1\ [r]\\ [r]\ e_2\ [q]\end{array}}{[p]\ e_1 + e_2\ [q]}$$

Subtraction:

$$\frac{\begin{array}{c}[p]\ e_1\ [r]\\ [r]\ e_2\ [q]\end{array}}{[p]\ e_1 - e_2\ [q]}$$

Multiplication:

$$\frac{\begin{array}{c}[p]\ e_1\ [r]\\ [r]\ e_2\ [q]\end{array}}{[p]\ e_1 * e_2\ [q]}$$

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

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

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

6.1.2.1 Semantics of Numeric Expression Specification

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

$$\begin{array}{ll} \textit{Null list:} & \textit{Cons:} \\ \dfrac{}{[q]\ \langle\ \rangle\ [q]} & \dfrac{\begin{array}{c}[p]\ e\ [r] \\ [r]\ es\ [q]\end{array}}{[p]\ CONS\ e\ es\ [q]} \end{array}$$

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:

$$\frac{\begin{array}{c}[p]\ e_1\ [r]\\ {[r]}\ e_2\ [q]\end{array}}{[p]\ e_1 = e_2\ [q]}$$

Less Than:

$$\frac{\begin{array}{c}[p]\ e_1\ [r]\\ {[r]}\ e_2\ [q]\end{array}}{[p]\ e_1 < e_2\ [q]}$$

Lexicographic Less Than:

$$\frac{\begin{array}{c}[p]\ es_1\ [r]\\ {[r]}\ es_2\ [q]\end{array}}{[p]\ es_1 \ll es_2\ [q]}$$

Conjunction:

$$\frac{\begin{array}{c}[p]\ b_1\ [r]\\ {[r]}\ b_2\ [q]\end{array}}{[p]\ b_1 \wedge b_2\ [q]}$$

Disjunction:

$$\frac{\begin{array}{c}[p]\ b_1\ [r]\\ {[r]}\ b_2\ [q]\end{array}}{[p]\ b_1 \vee b_2\ [q]}$$

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:

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

Iteration:

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

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:

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

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 6.7: 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:

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

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

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

a_1	:	precondition
c	:	command
p	:	procedure name
a_2	:	entrance condition
ρ	:	procedure environment

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:

$$\frac{\begin{array}{c}\{a_0 \Rightarrow a_1\}\\ \{a_1\}\ c \rightarrow p\ \{a_2\}\ /\rho\end{array}}{\{a_0\}\ c \rightarrow p\ \{a_2\}\ /\rho}$$

Entrance Condition Weakening:

$$\frac{\begin{array}{c}\{a_1\}\ c \rightarrow p\ \{a_2\}\ /\rho\\ \{a_2 \Rightarrow a_3\}\end{array}}{\{a_1\}\ c \rightarrow p\ \{a_3\}\ /\rho}$$

Entrance Condition Conjunction:

$$\frac{\begin{array}{c}\{a_1\}\ c \rightarrow p\ \{a_2\}\ /\rho\\ \{a_1\}\ c \rightarrow p\ \{a_3\}\ /\rho\end{array}}{\{a_1\}\ c \rightarrow p\ \{a_2 \wedge a_3\}\ /\rho}$$

False Precondition:

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

Skip:

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

Abort:

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

Assignment:

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

Sequence:

$$\frac{\begin{array}{c}\{a_1\}\ c_1 \rightarrow p\ \{q\}\ /\rho\\ \{a_1\}\ c_1\ \{a_2\}\ /\rho\\ \{a_2\}\ c_2 \rightarrow p\ \{q\}\ /\rho\end{array}}{\{a_1\}\ c_1\ ;\ c_2 \rightarrow p\ \{q\}\ /\rho}$$

Conditional:

$$\frac{\begin{array}{c}\{a_1\}\ c_1 \rightarrow p\ \{q\}\ /\rho\\ \{a_2\}\ c_2 \rightarrow p\ \{q\}\ /\rho\end{array}}{\begin{array}{l}\{AB\ b \Longrightarrow ab\_pre\ b\ a_1 \mid ab\_pre\ b\ a_2\}\\ \quad \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \rightarrow p\ \{q\}\ /\rho\end{array}}$$

Iteration:

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

Procedure Call:

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

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

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** $n < \widehat{n}$ 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

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

The substitutions evaluate as

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

Then the call progress claim is proven as follows.

```
1. ${(n-1) < \hat{n}}$ $even(a; n-1) \rightarrow even$ $\{n < \hat{n}\}$ $/\rho$      Procedure Call
                                                                                Rule (2nd)

2. $\{\textbf{true}\}$ $odd(a; n-2) \rightarrow even$ $\{n < \hat{n}\}$ $/\rho$        Procedure Call
                                                                                Rule (1st)

3.  $\{n = 1 => (n-1) < \hat{n} \mid \textbf{true}\}$                       1, 2,  Conditional
     if $n = 1$  then $even(a; n-1)$                                              Rule
                 else  $odd(a; n-2)$
     fi
    $\rightarrow even$ $\{n < \hat{n}\}$ $/\rho$

4. $\{\textbf{true}\}$ $a := 0 \rightarrow even$ $\{n < \hat{n}\}$ $/\rho$                   Assignment Rule

5.  $\{n = 0 => \textbf{true}$                                              4, 3,  Conditional
              $\mid (n = 1 => (n-1) < \hat{n} \mid \textbf{true})\}$                Rule
    if $n = 0$ then $a := 0$
    else if $n = 1$  then $even(a; n-1)$
                     else  $odd(a; n-2)$
         fi
    fi
    $\rightarrow even$ $\{n < \hat{n}\}$ $/\rho$

6. $\{\hat{n} = n \Rightarrow$  $(n = 0 => \textbf{true}$                            Tautology
                   $\mid (n = 1 => (n-1) < \hat{n} \mid \textbf{true})$ $)\}$

7.  $\{\hat{n} = n\}$                                                       6, 5,  Precondition
    if $n = 0$ then $a := 0$                                                     Strengthening
    else if $n = 1$  then $even(a; n-1)$
                     else  $odd(a; n-2)$
         fi
    fi
    $\rightarrow even$ $\{n < \hat{n}\}$ $/\rho$
```

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

calls *odd* **with** $n < \hat{n}$

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:

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

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

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

a_1	:	precondition
p_1	:	starting procedure name
ps	:	path (list of procedure names)
p_2	:	destination procedure name
a_2	:	entrance condition
ρ	:	procedure environment

6.3.4.1 Semantics of Path Entrance Specification

$$\begin{aligned}&\{a_1\}\ p_1 \text{ — } ps \rightarrow p_2\ \{a_2\}\ \rho\ =\\ &\qquad (\forall s_1\ s_2.\ A\ a_1\ s_1 \wedge M\_calls\ p_1\ s_1\ ps\ p_2\ s_2\ \rho \Rightarrow A\ a_2\ s_2)\end{aligned}$$

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):

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

Transitivity:

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

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.

$$\frac{\begin{array}{c} WF_{env\_pre}\ \rho \\ \rho\ p_1 = \langle vars, vals, glbs, pre, post, calls, rec, c\rangle \\ \rho\ p_2 = \langle vars', vals', glbs', pre', post', calls', rec', c'\rangle \end{array}}{\{pre\}\ p_1 — ps \rightarrow p_2\ \{pre'\}\ /\rho}$$

Precondition Strengthening:

$$\frac{\begin{array}{c}\{a_0 \Rightarrow a_1\} \\ \{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_2\}\ /\rho\end{array}}{\{a_0\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_2\}\ /\rho}$$

Entrance Condition Conjunction:

$$\frac{\begin{array}{c}\{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_2\}\ /\rho \\ \{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_3\}\ /\rho\end{array}}{\{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_2 \wedge a_3\}\ /\rho}$$

Entrance Condition Weakening:

$$\frac{\begin{array}{c}\{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_2\}\ /\rho \\ \{a_2 \Rightarrow a_3\}\end{array}}{\{a_1\}\ p_1 \text{ --- } ps \rightarrow p_2\ \{a_3\}\ /\rho}$$

False Precondition:

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

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:

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

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

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

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

The invocation of *call_progress* evaluates as

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

Thus we have proven

$$\{\forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \widehat{n})\}\ \mathit{odd} \;\text{---}\; \langle\rangle \rightarrow \mathit{even}\ \{n < \widehat{n}\}\ /\rho$$

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

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

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

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

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.

$$\boxed{\begin{array}{l} \mathit{call\_path\_progress}\ p_1\ \langle\ \rangle\ p_2\ q\ \rho\ = \\ \qquad\qquad \mathit{call\_progress}\ p_1\ p_2\ q\ \rho \\ \\ \mathit{call\_path\_progress}\ p_1\ (\mathit{CONS}\ p\ ps)\ p_2\ q\ \rho\ = \\ \qquad\qquad \mathit{call\_progress}\ p_1\ p\ (\mathit{call\_path\_progress}\ p\ ps\ p_2\ q\ \rho)\ \rho \end{array}}$$

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:

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

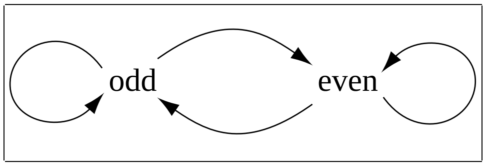

Figure 6.2: Procedure Call Graph for Odd/Even Example.

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* → *odd* → *even*. By the Call Path Progress Rule given above, we have

$$\begin{array}{l} \{\textbf{true} \wedge call\_path\_progress\ odd\ \langle odd\rangle\ even\ (n < \hat{n})\ \rho\} \\ \qquad\qquad odd - \langle odd\rangle \rightarrow even\ \{n < \hat{n}\}\ /\rho \end{array}$$

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

$$\begin{aligned} &\mathit{call\_progress}\ \mathit{odd}\ \mathit{even}\ (n < \hat{n})\ \rho \\ &\qquad = \forall a_1, n_1.\ (n_1 < n) \Rightarrow (n_1 < \hat{n}) \end{aligned}$$

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

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

Thus we have proven

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

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

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

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

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

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

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:

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

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:

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

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:

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

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:

$$\frac{\begin{array}{l}\{a_0 \Rightarrow a_1\}\\ [a_1]\ c \downarrow /\rho\end{array}}{[a_0]\ c \downarrow /\rho}$$

False Precondition:

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

Skip:

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

Abort:

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

Assignment:

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

Sequence:

$$\frac{\begin{array}{c}[a_1]\ c_1 \downarrow /\rho\\ \{a_1\}\ c_1\ \{a_2\}\ /\rho\\ [a_2]\ c_2 \downarrow /\rho\end{array}}{[a_1]\ c_1\ ;\ c_2 \downarrow /\rho}$$

Conditional:

$$\frac{\begin{array}{c}[a_1]\ c_1 \downarrow /\rho\\ [a_2]\ c_2 \downarrow /\rho\end{array}}{\begin{array}{c}[AB\ b \Rightarrow ab\_pre\ b\ a_1 \mid ab\_pre\ b\ a_2]\\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{fi} \downarrow /\rho\end{array}}$$

Iteration:

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

Procedure Call:

$$\frac{\begin{array}{c}WF_{env\_syntax}\ \rho\\ WF_c\ (\mathbf{call}\ p(xs;es))\ g\ \rho\end{array}}{[a]\ \mathbf{call}\ p(xs;es) \downarrow /\rho}$$

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:

$$\begin{array}{l} \{p \Rightarrow q\} \\ \underline{[q]\ c \Downarrow /\rho} \\ [p]\ c \Downarrow /\rho \end{array}$$

False Precondition:

$$\overline{[\mathbf{false}]\ c \Downarrow /\rho}$$

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:

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

Skip:

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

Abort:

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

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:

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

Iteration:

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

Rule of Adaptation:

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

Procedure Call:

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

Table 6.15: Hoare Logic for Command Termination.

Precondition Strengthening:

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

Postcondition Weakening:

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

False Precondition:

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

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:

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

Abort:

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

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:

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

Iteration:

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

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:

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

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:

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

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

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