Trustworthy Tools for Trustworthy Programs: Automatic Verification of Mutually Recursive Procedures

Peter V. Homeier and David F. Martin

Computer Science Department
University of California, Los Angeles
homeier@cs.ucla.edu and dmartin@cs.ucla.edu

Abstract. Verification Condition Generator (VCG) tools have been effective in simplifying the task of proving programs correct. However, in the past these VCG tools have in general not themselves been mechanically proven, so any proof using and depending on these VCGs might have contained errors. In our work, we define and rigorously prove correct a VCG tool within the HOL theorem proving system, for a standard imperative language, notably containing mutually recursive procedures and expressions with side effects. Starting from a structural operational semantics of this programming language, we prove as theorems the axioms and rules of inference of a Hoare-style axiomatic semantics, verifying their soundness. This axiomatic semantics is then used to define and prove correct a VCG tool for this language. Finally, this verified VCG is applied to example programs to verify their partial correctness.

1 Introduction

The most common technique used today to produce quality software without errors is testing. However, even repeated testing cannot reliably eliminate all errors, and hence is incomplete. To achieve a higher level of reliability and trust, programmers may construct proofs of correctness, verifying that the program satisfies a formal specification. This need be done only once, and eliminates whole classes of errors. However, these proofs are complex, full of details, and difficult to construct by hand, and thus may themselves contain errors, which reduces trust in the program so proved. Mechanical proofs are more secure, but even more detailed and difficult.

One solution to this difficulty is partially automating the construction of the proof by a tool called a *Verification Condition Generator* (VCG). This VCG tool writes the proof of the program, modulo a set of formulas called *verification conditions* which are left to the programmer to prove. These verification conditions do not contain any references to programming language phrases, but only deal with the logics of the underlying data types. This twice simplifies the programmer's burden, reducing the volume of proof and level of proof, and makes the process more effective. However, in the past these VCG tools have not in general themselves been proven, meaning that the trust of a program's proof rested on the trust of an unproven VCG tool.

In this work we define a VCG within the Higher Order Logic (HOL) theorem proving system [6], for a programming language with mutually recursive procedures, and prove that the truth of the verification conditions it returns suffice to verify the asserted program submitted to the VCG. This theorem stating the VCG's correctness then supports the use of the VCG in proving the correctness of individual programs with complete soundness assured. The VCG automates much of the work and detail involved, relieving the programmer of all but the essential task of proving the verification conditions. This enables proofs of programs which are both effective and trustworthy to a degree not previously seen together.

2 Previous Work

There has been very little work done on proving the correctness of expressions; an exception is Sokolowski's paper on a "term-wise" approach to partial correctness [11]. Even he does not treat expressions with side effects. Side effects appear commonly in "real" programming languages, such as in 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.

Several authors have treated recursive procedures, varying in the flexibility or rigidity of the proof techniques, as well as in the expressive power of the procedures themselves. The passing of parameters has been a key area. Some proposals have later been found unsound. This highlights the essential subtlety of this area.

In this paper, we define a "verified" verification condition generator as one which has been proven to correctly produce, for any input program and specification, a set of verification conditions whose truth implies the consistency of the program with its specification. Preferably, this verification of the VCG will be mechanically checked for soundness, because of the many details and deep issues that arise. Many VCG's have been written but not verified; there is then no assurance that the verification conditions produced are properly related to the original program, and

hence no security that after proving the verification conditions, the correctness of the program follows. Gordon's work below is an exception in that the security is maintained by the HOL system itself.

Igarashi, London, and Luckham in 1973 gave an axiomatic semantics for a subset of Pascal including procedures, and described a VCG they had written in MLISP2 [8]. 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.

Larry Ragland, also in 1973, verified a verification condition generator written in Nucleus, a language Ragland invented to both express a VCG and be verifiable [10]. 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 generator used an unverified VCG written in Snobol4. The verification conditions it generated were proven by hand, not mechanically. This proof substantially increased the degree of trustworthiness of Ragland's VCG.

Michael Gordon in 1989 did the original work of constructing within HOL a framework for proving the correctness of programs [5]. This work did not cover procedures. Gordon 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. The 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. If it succeeded, the resulting subgoals were soundly related to the original correctness goal by the security of HOL itself. Fundamentally, there were certain limitations to the expressive power and proven conclusions of this approach, as recognized by Gordon himself:

> "$\mathcal{P}[\mathcal{E}/\mathcal{V}]$ (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:
>
> $\vdash \forall P\ E\ V.$ **Spec** (**Truth**(**Subst**(*P, E, V*)), **Assign**(*V,* **Value** *E*), **Truth** *P*)
>
> It is clear that working out the details of this would be a lot of work." [5]

In 1991, Sten Agerholm [1] 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 additional information to handle sequences of commands and the **while** command, to be supplied by the user.

This paper explores the alternative approach described but not investigated by Gordon. It turns out to yield great expressiveness and control in stating and proving as theorems within HOL concepts which previously were only describable as meta-theorems outside HOL, as above. For example, we are able to prove the assignment axiom that Gordon cannot:

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

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 semantic 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 [13]. 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 paper, where repeatedly a syntactic manipulation is defined, and then its semantics is described and proven correct within HOL.

Our previous paper [7] treated partial correctness of a standard while-loop language, including the unusual feature of expressions with side effects, but without procedures. We extend this work here to cover the partial correctness of systems of mutually recursive procedures. Many new concepts must be introduced, for example, programs must be checked for well-formedness before their execution or verification. This is a test that need be performed once, for example at compile time, as a static check. In the language being studied, this includes checking

that each procedure is called with the right number of parameters, and that each procedure declaration gives a specification of its behavior consistent with its actual code body, as well as a number of conditions on the proper use of variables and parameters. An interesting feature of this system is that the recursive proof inherent in using mutually recursive procedures is resolved once for all programs, leaving only a set of non-recursive verification conditions for the programmer to prove.

3 Higher Order Logic

Higher Order Logic (HOL) [6] is a version of predicate calculus that allows variables to range over functions and predicates. Thus denotable values may be functions of any higher order. Strong typing ensures the consistency and proper meaning of all expressions. The power of this logic is similar to set theory, and it is sufficient for expressing most mathematical theories.

HOL is also a mechanical proof development system. It is secure in that only true theorems can be proved. Rather than attempting to automatically prove theorems, HOL acts as a supportive assistant, mechanically checking the validity of each step attempted by the user.

The primary interface to HOL is the polymorphic functional programming language ML ("Meta Language") [4]; commands to HOL are expressions in ML. Within ML is a second language OL ("Object Language"), representing terms and theorems by ML abstract datatypes **term** and **thm**. A shallow embedding represents program constructs by new OL functions to combine the semantics of the constituents to produce the semantics of the combination. Our approach is to define a *third* level of language, contained within OL as concrete recursive datatypes, to represent the constructs of the programming language PL being studied and its associated assertion language AL. We begin with the definition of variables.

4 Variables and Variants

A variable is represented by a new concrete type `var`, with one constructor, VAR `:string->num->var`. We define two deconstructor functions, Base(VAR *str n*) = *str* and Index(VAR *str n*) = *n*. 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. Some languages distinguish between program variables and logical variables, which cannot be changed by program control. In this language, we denote logical variables by beginning its name with a carat (^) 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->(var)set->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. This 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* | *x*) (*)

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 HOL's recursive function definition operator. As a substitute, we wanted to define the *variant* function using `new_specification` by specifying its properties, as

1) (*variant x s*) *is_variant x*, and
2) ~(*variant x s* IN *s*), and
3) ∀*z*. if (*z is_variant x*) ∧ ~(*z* IN *s*), then Index(*variant x s*) ≤ Index(*z*),

where *y is_variant x* = (Base(*y*) = Base(*x*) ∧ Index(*x*) ≤ 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 third property's antecedent is infinite, and we were working strictly with finite sets. The solution was to introduce the function *variant_set*, where *variant_set x n* returns the set of the first *n* variants of *x*, all different from each other, so CARD (*variant_set x n*) = *n*. The definition of *variant_set* is

variant_set x 0 = EMPTY ∧
variant_set x (SUC *n*) = (*mk_variant x n*) INSERT (*variant_set x n*).

Then by the pigeonhole principle, we are guaranteed that there must be at least one variable in *variant_set x* (SUC (CARD *s*)) which is not in the set *s*. This leads to the needed existence theorem. We then defined *variant* with the following properties:

1') (*variant x s*) IN *variant_set x* (SUC (CARD *s*)), and
2') ~(*variant x s* IN *s*), and
3') ∀z.if *z* IN *variant_set x* (SUC (CARD *s*)) ∧ ~(*z* IN *s*),
then Index(*variant x s*) ≤ Index(*z*).

From this definition, we then proved both the original set of properties (1)–(3), and also the constructive function definition given above (*), as theorems.

Finally, given the definition of *variant*, we define a similar operator on lists:

variants [] *s* = [] ∧
variants (CONS *x xs*) *s* = **let** *x'* = *variant x s* **in**
CONS *x'* (*variants xs* (*x'* INSERT *s*)).

This definition has the property that the resulting list has no duplicates.

5 Programming and Assertion Languages

The syntax of the programming language PL is

`exp:`	e	::=	$n \mid x \mid {+}{+}x \mid e_1 + e_2 \mid e_1 - e_2$
`bexp:`	b	::=	$e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b$
`cmd:`	c	::=	**skip** | **abort** | $x := e$ | $c_1; c_2$ | **if** b **then** c_1 **else** c_2 | **assert** a **while** b **do** c | $p(es)$
`decl:`	d	::=	**procedure** p(**val** $x_1, \ldots, x_n$); **global** $y_1, \ldots, y_n$; **pre** a_1; **post** a_2; c **end procedure** | $d_1; d_2$ (this will be represented later as **proc** *p vs glbs pre post c*)
`prog:`	π	::=	**program** d ; c **end program**

Table 1. Programming Language Syntax

Most of these constructs are standard. *n* is an unsigned integer; *x* and *y* are program variables, required not to begin with the character "^"; these names are reserved as "logical" variables. ++ is the increment operator; **abort** causes an immediate abnormal termination; the **while** loop requires an invariant assertion to be supplied. In the procedure call *p*(*es*), *p* is simply a string, while *es* is a list of `exp` expressions, denoting actual value parameters.

The procedure declaration specifies the procedure's name *p*, value formal parameter names $x_1, \ldots, x_n$, global variables used in *p* (or any procedure *p* calls) $y_1, \ldots, y_n$, precondition a_1, postcondition a_2, and body *c*. All parameter types are `num`. Procedures are mutually recursive, and may call each other irrespective of their order. If two procedures have the same name, the latter prevails.

The notation used above is for ease of reading; each phrase is actually formed by a constructor function, e.g., `PROC:string->(var)list->(var)list->aexp->aexp->cmd->decl` for procedure declaration. In line with this constructor `PROC`, we will refer to a typical procedure declaration as **proc** *p vs glbs pre post c*, instead of the longer version given above. We overload the same operator in different languages, asking the reader to disambiguate by context.

The syntax of the associated assertion language AL is

`vexp:`	v	::=	$n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$
`aexp:`	a	::=	**true** | **false** | $v_1 = v_2 \mid v_1 < v_2 \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid \sim a \mid a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid a_1 \Rightarrow a_2 \mid a_3$ | **close** a | $\forall x.\ a$ | $\exists x.\ a$

Table 2. Assertion Language Syntax

Again, most of these expressions are standard. $a_1 \Rightarrow a_2 \mid a_3$ is a conditional expression, yielding the value of a_2 or a_3 depending on the value of a_1. **close** *a* forms the universal closure of *a*, which is true when *a* is true for all possible assignments to its free variables. Again, the notation is for readability; each phrase is actually created by a constructor function, e.g., the constructor `AVAR:var->vexp` creates a `vexp` from a variable.

6 Operational Semantics

We define the type `state` as `var->num`, a mapping from variables to numbers. We also define the type `env` as `string->((var)list # (var)list # aexp # aexp # cmd)`, representing a family of procedure declarations, indexed by the name of the procedure. The elements of the tuple are, in order, the value parameter list, the globals list, the precondition, the postcondition, and the body.

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

$E\ e\ s_1\ n\ s_2$:	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`:env` and state s_1 yields state s_2.
$D\ d\ \rho_1\ \rho_2$:	declaration d`:decl` elaborated in environment ρ_1`:env` yields result environment ρ_2.
$P\ \pi\ s$:	program π`:prog` executed yields state s.

Here is the structural operational semantics [13] of the programming language PL, given as rules inductively defining the six relations *E*, *ES*, *B*, *C*, *D*, and *P*. These relations (except for *ES*) are defined within HOL using Tom Melham's excellent rule induction package [2,9]. The notation $f[e/x]$ indicates the function f updated so that

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

For defining *P*, we define ρ_0 as the empty environment $\rho_0 = \lambda p.\langle [], [],$ **false**, **true**, **abort**$\rangle$, and s_0 as the initial state $s_0 = \lambda x.\ 0$. We may construct an environment ρ from a declaration d as $\rho = mkenv\ d\ \rho_0$, where

$$\begin{aligned} mkenv\ (\textbf{proc}\ p\ vs\ glbs\ pre\ post\ c)\ \rho &= \rho[(vs, glbs, pre, post, c)\ /\ p] \\ mkenv\ (d_1; d_2)\ \rho &= mkenv\ d_2\ (mkenv\ d_1\ \rho) \end{aligned}$$

E

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:

$$\frac{E\ e_1\ s_1\ n_1\ s_2, \quad E\ e_2\ s_2\ n_2\ s_3}{E\ (e_1 + e_2)\ s_1\ (n_1 + n_2)\ s_3}$$

Subtraction:

$$\frac{E\ e_1\ s_1\ n_1\ s_2, \quad E\ e_2\ s_2\ n_2\ s_3}{E\ (e_1 - e_2)\ s_1\ (n_1 - n_2)\ s_3}$$

ES

Empty List:

$$\frac{}{ES\ \textbf{nil}\ s\ \textbf{nil}\ s}$$

Cons List:

$$\frac{E\ e\ s_1\ n\ s_2, \quad ES\ es\ s_2\ ns\ s_3}{ES\ (\textbf{cons}\ e\ es)\ s_1\ (\textbf{cons}\ n\ ns)\ s_3}$$

B

Equality:

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

Less Than:

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

Conjunction:

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

Disjunction:

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

Negation:

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

Table 3. Programming Language Structural Operational Semantics

C	*Skip:* $\dfrac{}{C\ \textbf{skip}\ \rho\ s\ s}$ *Abort:* (no rules) *Assignment:* $\dfrac{E\ (e)\ s_1\ n\ s_2}{C\ (x := e)\ \rho\ s_1\ s_2[n/x]}$ *Sequence:* $\dfrac{C\ c_1\ \rho\ s_1\ s_2, \quad C\ c_2\ \rho\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ \rho\ s_1\ s_3}$ *Conditional:* $\dfrac{B\ b\ s_1\ \mathrm{T}\ s_2, \quad C\ c_1\ \rho\ s_2\ s_3}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ \rho\ s_1\ s_3}$ $\dfrac{B\ b\ s_1\ \mathrm{F}\ s_2, \quad C\ c_2\ \rho\ s_2\ s_3}{C\ (\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2)\ \rho\ s_1\ s_3}$ *Iteration:* $\dfrac{B\ b\ s_1\ \mathrm{T}\ s_2, \quad C\ c\ \rho\ s_2\ s_3 \quad C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c)\ \rho\ s_3\ s_4}{C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c)\ \rho\ s_1\ s_4}$ $\dfrac{B\ b\ s_1\ \mathrm{F}\ s_2}{C\ (\textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c)\ \rho\ s_1\ s_2}$ *Call:* $\dfrac{ES\ es\ s_1\ ns\ s_2 \quad \rho(p) = \langle vs, glbs, pre, post, c\rangle \quad C\ c\ \rho\ s_2[ns/vs]\ s_3}{C\ (\textbf{call}\ p(es))\ \rho\ s_1\ s_2[\ (\textbf{map}\ s_3\ glbs)\ /glbs]}$
D	*Procedure Declaration:* $\dfrac{}{D\ (\ \textbf{proc}\ vs\ glbs\ pre\ post\ c)\ \rho\ \ \rho[\langle vs, glbs, pre, post, c\rangle / p]}$ *Declaration Sequence:* $\dfrac{D\ d_1\ \rho_1\ \rho_2, \quad D\ d_2\ \rho_2\ \rho_3}{D\ (d_1;\ d_2)\ \rho_1\ \rho_3}$
P	*Program:* $\dfrac{D\ d\ \rho_0\ \rho_1, \quad C\ c\ \rho_1\ s_0\ s_1}{P\ (\textbf{program}\ d;\ c\ \textbf{end program})\ s_1}$

Table 3. Programming Language Structural Operational Semantics (Continued)

The semantics of the assertion language AL is given by recursive functions defined on the structure of the construct, in a directly denotational fashion:

V v s: numeric expression *v* : `vexp` evaluated in state *s*, yields a numeric value in `num`. (*VS vs s* is the same for lists.)
A a s: boolean expression *a* : `aexp` evaluated in state *s*, yields a truth value in `bool`.

V	$V\ n\ s = n$ $V\ x\ s = s(x)$ $V\ (v_1 + v_2)\ s = V\ v_1\ s + V\ v_2\ s$ (–, * treated analogously)
A	A **true** $s = \mathrm{T}$ A **false** $s = \mathrm{F}$ $A\ (v_1 = v_2)\ s = (V\ v_1\ s = V\ v_2\ s)$ (< treated analogously) $A\ (a_1 \wedge a_2)\ s = (A\ a_1\ s \wedge A\ a_2\ s)$ ($\vee$, ~, $\Rightarrow$, a_1=a_2, a_1=>a_2|a_3 treated analogously) A (**close** a) $s = (\forall s_1.\ A\ a\ s_1)$ $A\ (\forall x.\ a)\ s = (\forall n.\ A\ a\ s[n/x])$ $A\ (\exists x.\ a)\ s = (\exists n.\ A\ a\ s[n/x])$

Table 4. Assertion Language Denotational Semantics

7 Substitution

7.1. Simultaneous substitution

We define proper substitution on assertion language expressions using the technique of *simultaneous substitutions*, following Stoughton [12]. 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->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:

$$[v/x] = \lambda y.(y{=}x \Rightarrow v \mid \mathsf{AVAR}\ y).$$

We apply a substitution by the infix operator $\lhd$. Thus, $a \lhd ss$ denotes the application of the simultaneous substitution *ss* to the expression *a*, where *a* can be either `vexp` or `aexp`. Therefore $a \lhd [v/x]$ denotes the single substitution of the expression *v* for the variable *x* wherever *x* appears free in *a*. 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 $s \lhd ss = \lambda y.(V\ (ss\ y)\ s)$.

Most of the cases of the definition of the application of a substitution to an expression are simply the distribution of the substitution across the immediate subexpressions. For example, the application of a substitution to a conjunction is

$$(a_1 \wedge a_2) \lhd ss = (a_1 \lhd ss) \wedge (a_2 \lhd ss)$$

The interesting cases of the definition of $a \lhd ss$ are where *a* is a quantified expression, e.g.:

$$\begin{aligned}(\forall x.\ a) \lhd ss = \quad & \textbf{let } free = \bigcup_{z \in (FV_a\ a) - \{x\}} FV_v\ (ss\ z) \quad \textbf{in} \\ & \textbf{let } y = variant\ x\ free \textbf{ in} \\ & \forall\ y.\ a \lhd (ss[(\mathsf{AVAR}\ y)\ /\ x])\end{aligned}$$

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 following two theorems about the semantics of substitution:

$$\vdash \forall v\ s\ ss.\ V(v \lhd ss)\ s = V\ v\ (s \lhd ss)$$

$$\vdash \forall a\ s\ ss.\ A(a \lhd ss)\ s = A\ a\ (s \lhd ss)$$

This is our statement of the Substitution Lemma of logic, and essentially says that syntactic substitution is equivalent to semantic substitution.

7.2. Variable substitution

The substitutions discussed above replaced variables by 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->var`. The application of these substitutions to assertion expressions is defined similarly that described above; most of the cases are the distribution of the substitution across the immediate subexpressions. Notably, the definition of substitution on quantified expressions is different in that the bound variables are not protected, but in fact are included, in the transformation. Thus, for *vss*`:vsubst`:

$$(\forall x.\ a) \lhd vss = \forall\ (vss\ x).\ (a \lhd vss)$$

The most common variable substitutions we will want to use will replace one list of variables by another (of equal length). The identity substitution is the identity function, ι_v`:var->var`. To create these variable substitutions, we introduce the operator $/_v$, to distinguish it from the / above:

$$[\ []\ /_v\ xs] = \iota_v$$

$$[ys\ /_v\ []\ ] = \iota_v$$

$$\begin{aligned}[(\text{CONS}\ y\ ys)\ /_v\ (\text{CONS}\ x\ xs)] = \textbf{let } vss = [ys\ /_v\ xs] \textbf{ in} \\ vss[(vss\ x)\ /\ (@z.\ vss\ z = y)]\ [y\ /\ x]\end{aligned}$$

where the `vsubst` *vss*, which is a mapping, is updated, binding first ($@\ z.\ vss\ z = y$) to $vss\ x$, and then x to y. @ here is the Hilbert selection operator. 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 invertible.

The virtue of these new `vsubst` substitutions is that we may apply them not only to assertion expressions, but also to program code, distributing their effect down into subexpressions until plain variables are reached, and then replacing them by the new variable found by applying the substitution function to the old variable. This is important in defining the adaptation of procedure specifications to situations where some of the local variables must be changed. The capability to consistently substitute variables across the body of a procedure and its specification will turn out to be vital in proving the general rule of procedure call. From the definitions given here, we may prove the semantics of such substitutions, as, for example for boolean program expressions (here o is functional composition):

$$\vdash\ \forall b\ s_1\ t\ s_2\ ys\ xs.\ B\ (b \lhd [ys\ /_v\ xs])\ s_1\ t\ s_2 = B\ b\ (s_1 \circ [ys\ /_v\ xs])\ t\ (s_2 \circ [ys\ /_v\ xs])$$

8 Translation

Expressions have typically not been treated in previous work on verification; there are some exceptions, notably Sokolowski [11]. 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 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`: exp -> subst -> (aexp # subst):`

$$\begin{array}{lcl}
VE1\ (n)\ ss & = & n, ss \qquad \textit{(where comma (,) makes a pair)} \\
VE1\ (x)\ ss & = & ss\ x, ss \\
VE1\ (++x)\ ss & = & (ss\ x) + 1, ss[((ss\ x) + 1) / x] \\
VE1\ (e_1 + e_2)\ ss & = & (VE1\ e_1 \rightarrow \lambda v_1.\ (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 + v_2, ss_2)))\ ss \\
VE1\ (e_1 - e_2)\ ss & = & (VE1\ e_1 \rightarrow \lambda v_1.\ (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 - v_2, ss_2)))\ ss
\end{array}$$

where $\rightarrow$ is a "translator continuation" operator, defined as

$$(f \rightarrow k)\ ss = \textbf{let}\ (v, ss') = f\ ss\ \textbf{in}\ k\ v\ ss'$$

Then define

$$\begin{array}{lcl}
VE\ e & = & \text{fst}\ (\ VE1\ e\ \iota\ ) \quad \textit{(where } \iota \textit{ is the identity substitution} \\
VE_state\ e & = & \text{snd}(\ VE1\ e\ \iota\ ) \quad \textit{and } \text{fst} \textit{ and } \text{snd} \textit{ select the members of a pair)}
\end{array}$$

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem:

$$\vdash\ \forall e\ s_1\ n\ s_2.\ (E\ e\ s_1\ n\ s_2) = (\ n = V\ (VE\ e)\ s_1 \ \wedge\ s_2 = s_1 \lhd (VE\_state\ e)\ )$$

A similar set of functions are used to translate boolean expressions. We define the helper functions *VES1*, *AB1* and the main translation functions *VES*, *VES_state*, *AB* and *AB_state*, and prove their correctness as

$$\vdash\ \forall es\ s_1\ ns\ s_2.\ (ES\ es\ s_1\ ns\ s_2) = (\ ns = VS\ (VES\ es)\ s_1 \ \wedge\ s_2 = s_1 \lhd (VES\_state\ es)\ )$$

$$\vdash\ \forall b\ s_1\ t\ s_2.\ (B\ b\ s_1\ t\ s_2) = (\ t = A\ (AB\ b)\ s_1 \ \wedge\ s_2 = s_1 \lhd (AB\_state\ b)\ )$$

These theorems mean that every evaluation of a programming language expression has its semantics completely captured by the two translation functions for its type. These are essentially small compiler correctness proofs.

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 assignment:

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

9 Well-Formedness

We define the semantics of Floyd/Hoare partial correctness formulae as follows:

`aexp:`	$\{a\}$	= **close** a	(the universal closure of a)
		= $\forall s.\ A\ a\ s$	(a is true in all states)
`exp:`	$\{p\}e\{q\}$	= $\forall p\ q\ e\ n\ s_1\ s_2.\ A\ p\ s_1 \wedge E\ e\ s_1\ n\ s_2 \Rightarrow A\ q\ s_2$	
`bexp:`	$\{p\}b\{q\}$	= $\forall p\ q\ b\ t\ s_1\ s_2.\ A\ p\ s_1 \wedge B\ b\ s_1\ t\ s_2 \Rightarrow A\ q\ s_2$	
`cmd:`	$\{p\}c\{q\}/\rho$	= $\forall p\ q\ c\ s_1\ s_2.\ A\ p\ s_1 \wedge C\ c\ \rho\ s_1\ s_2 \Rightarrow A\ q\ s_2$	

Table 5. Floyd/Hoare Partial Correctness Formulae Semantics

In specifying the behavior of a procedure, we require the programer to provide a precondition, specifying a necessary condition at time of entry, and a postcondition, specifying the procedure's behavior as a resulting condition at time of exit. The postcondition must refer to the values of variables at both these times. To avoid ambiguity, we introduce *logical variables*.

Logical variables are barred from any mention within program code; they may not be assigned to, or even read in a program expression. Their only mention may be within assertion language expressions. Since they can never be assigned to, each must always have the same value unchanged throughout any execution of code. They may have different values from one execution to another, but in any one, they are fixed, and serve well to mark values of variables from a prior time in the execution. In a procedure's postcondition, logical variables denote the value of a variable at time of entry, and program variables denote the value of the variable at time of exit.

A logical variable is designated by a special initial character, for which we will use the caret (^) character. Normally variables are made of an arbitrary string (along with a number to assist in creating variants). But now we carve out a space of names for logical variables by restricting program variables from beginning with this special character. A state remains a mapping from all variables to their current integer values; since the logical and program variables have different names, there is no conflict.

This allows an easy generation of a logical variable from a program variable, by simply prefixing its name with caret (^); we define a function to do this, *logical* `:var->var`, and a similar function for lists of variables, *logicals* `:(var)list->(var)list`. In a procedure's postcondition, these logical variables are used to designate the *prior* value of a variable, the value it had at the time the procedure was entered; the program variable itself is used unchanged to designate the value of the variable at the time of exit.

Since these logical variables are syntactically the same type as program variables, there is a need for a test to ensure that logical variables do not appear in program text. This test is part of a static test, to be run before any execution or verification, called "well-formedness", which is defined by a function WF_φ for each kind of program phrase φ, where $\varphi \in \{s,v,e,es,b,c,d,p\}$, testing strings, variables, numeric expressions, lists of numeric expressions, boolean expressions, commands, declarations, and programs. It turns out that in addition to checking for the exclusion of logical variables, it is also needful for the well-formedness tests to perform other static checks as well, such as ensuring that a procedure call has the right number of arguments for its definition. In fact, we also need to check that an entire environment of procedures is correctly and consistently defined. This last is the well-formedness test for environments, evaluated by the function WF_{env}.

A string s is well-formed ($WF_s\ s$) if the first character is not "^":

WF_s '' = F
WF_s (STRING $a\ s$) = ~(a = LOG_CHAR), where LOG_CHAR = "^"

A variable v is well-formed ($WF_v\ v$) if its string is well-formed:

WF_v (VAR $s\ n$) = $WF_s\ s$

A numeric expression e is well-formed ($WF_e\ e$) if every part is well-formed:

$WF_e\ (n)$ = T
$WF_e\ (x) = WF_v\ x$
$WF_e\ (++x) = WF_v\ x$
$WF_e\ (e_1 + e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2$
$WF_e\ (e_1 - e_2) = WF_e\ e_1\ \wedge\ WF_e\ e_2$

Table 6. Well-Formedness Predicates

A list of numeric expressions *es* is well-formed (WF_{es} *es*) if every part is well-formed:

WF_{es} ([]) = T
WF_{es} (CONS *e es*) = WF_e *e* $\wedge$ WF_{es} *es*

A boolean expression *b* is well-formed (WF_b *b*) if every part is well-formed:

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 ($b_1 \wedge b_2$) = 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 (~*b*) = WF_b *b*

A command *c* is well-formed in an environment ρ (WF_c *c* ρ) if every part is well-formed, and if every call supplies the same number of actual parameters as the procedure has formal parameters:

WF_c (**skip**) ρ = T
WF_c (**abort**) ρ = T
WF_c (*x* := *e*) ρ = WF_v *x* $\wedge$ WF_e *e*
WF_c (c_1; c_2) ρ = WF_c c_1 ρ $\wedge$ WF_c c_2 ρ
WF_c (**if** *b* **then** c_1 **else** c_2) ρ = WF_b *b* $\wedge$ WF_c c_1 ρ $\wedge$ WF_c c_2 ρ
WF_c (**assert** *a* **while** *b* **do** *c*) ρ = WF_b *b* $\wedge$ WF_c *c* ρ
WF_c (*p*(*es*)) ρ = WF_{es} *es* $\wedge$ LENGTH((FST o ρ)*p*) = LENGTH(*es*)

A procedure specification <*vs*, *glbs*, *pre*, *post*, *c*> is *syntactically well-formed* in an environment ρ (WF_{proc_syntax} <*vs*, *glbs*, *pre*, *post*, *c*> ρ) *iff*

let *x* = *vs* & *glbs* (where "&" appends two lists), and let x_0 = *logicals x* in

1) *ALL_EL* WF_v *x* (every variable in *vs* and *glbs* is well-formed, i.e., not logical)
2) *DL x* ("disjoint list": *vs* and *glbs* have no duplicates within or between them)
3) WF_c *c* ρ (*c* is well-formed)
4) GV_c *c* $\rho \subseteq glbs$ (all globals referenced by procedures called within *c* are in *glbs*)
5) FV_c *c* $\rho \subseteq x$ (all free variables of *c* are in *x*)
6) FV_a *pre* $\subseteq x$ (all free variables of *pre* are in *x*)
7) FV_a *post* $\subseteq (x \cup x_0)$ (all free variables of *post* are in *x* or in x_0)

A declaration *d* is well-formed in an environment ρ (WF_d *d* ρ) if every individual procedure declaration is syntactically well-formed:

WF_d (**proc** *p vs glbs pre post c*) ρ = WF_{proc_syntax} <*vs*, *glbs*, *pre*, *post*, *c*> ρ
WF_d (d_1; d_2) ρ = WF_d d_1 ρ $\wedge$ WF_d d_2 ρ

A program π is well-formed (WF_p π) if both its declarations and its body are well-formed in the environment the declarations create:

WF_p (**program** *d* ; *c* **end program**) = **let** ρ = *mkenv d* ρ_0 **in** WF_d *d* ρ $\wedge$ WF_c *c* ρ

A procedure specification <*vs*, *glbs*, *pre*, *post*, *c*> is well-formed (both syntactically and semantically) (WF_{proc} <*vs*, *glbs*, *pre*, *post*, *c*> ρ) *iff*

let *x* = *vs* & *glbs* (where "&" appends two lists), and let x_0 = *logicals x* in

1) WF_{proc_syntax} <*vs*, *glbs*, *pre*, *post*, *c*> ρ (the specification is syntactically well-formed)

2) $\{x_0 = x \wedge pre\}\ c\ \{post\}\ /\rho$ (*c* is partially correct with respect to precondition ($x_0 = x \wedge$ *pre*) and postcondition *post* in environment ρ)

An environment ρ is well-formed ($WF_{env}(\rho)$) *iff*

$$\forall p.\ WF_{proc}\ (\rho\ p)\ \rho$$

Table 6. Well-Formedness Predicates (Continued)

This definition of the well-formedness of ρ may appear circular, in that ρ is used in the definition of both syntactic and semantic well-formedness. However, this circularity is resolved in that these tests may be evaluated

using only the "header" information of each procedure in ρ, i.e., without referring to their bodies.

Summarizing, for the most part, program constructs are well-formed (WF_v, WF_e, WF_{es}, WF_b, WF_c, WF_d, WF_p) if their constituent constructs are well-formed. The basic features checked are that

1) no logical variables are used in program text (outside assertion-language annotations);
2) procedure calls must agree with the environment in number of arguments;
3) procedure declarations d must satisfy syntactic well-formedness;
4) given syntactic well-formedness, environments ρ must satisfy partial correctness conditions. These will be established later via a set of verification conditions.

10 Axiomatic Semantics

We can now express the axiomatic semantics of the programming language, and *prove* each rule as a theorem from the previous structural operational semantics:

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:

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

Conditional:

$$\frac{\begin{array}{c}\{p \wedge AB(b)\}\ b\ \{r_1\} \\ \{p \wedge \sim AB(b)\}\ b\ \{r_2\} \\ \{r_1\}\ c_1\ \{q\}\ /\ \rho \\ \{r_2\}\ c_2\ \{q\}\ /\ \rho\end{array}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

Iteration:

$$\frac{\begin{array}{c}\{a \wedge AB(b)\}\ b\ \{p\} \\ \{a \wedge \sim AB(b)\}\ b\ \{q\} \\ \{p\}\ c\ \{a\}\ /\ \rho\end{array}}{\{a\}\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}\ /\ \rho}$$

Rule of Adaptation (if $x_0 \cap FV_a\ q = \emptyset$):

$$\frac{\begin{array}{c}WF_c\ c\ \rho, \quad WF_{env}\ \rho, \quad ALL\_EL\ WF_v\ x, \quad DL\ x \\ x_0 = logicals\ x, \quad x_0 \cap FV_a\ q = \emptyset \\ FV_c\ c\ \rho \subseteq x, \quad FV_a\ pre \subseteq x, \quad FV_a\ post \subseteq (x \cup x_0) \\ \{x_0 = x \wedge pre\}\ c\ \{post\}\ /\ \rho\end{array}}{\{pre \wedge ((\forall x.\ (post \Rightarrow q)) \lhd [x / x_0])\}\ c\ \{q\}\ /\ \rho}$$

Rule of Adaptation (general case):

$$\frac{\begin{array}{c}WF_c\ c\ \rho, \quad WF_{env}\ \rho, \quad ALL\_EL\ WF_v\ x, \quad DL\ x \\ x_0 = logicals\ x, \quad x'_0 = variants\ x_0\ (FV_a\ q) \\ FV_c\ c\ \rho \subseteq x, \quad FV_a\ pre \subseteq x, \quad FV_a\ post \subseteq (x \cup x_0) \\ \{x_0 = x \wedge pre\}\ c\ \{post\}\ /\ \rho\end{array}}{\{pre \wedge ((\forall x.\ (post \lhd [x'_0 / x_0] \Rightarrow q)) \lhd [x / x'_0])\}\ c\ \{q\}\ /\ \rho}$$

Table 7. Programming Language Axiomatic Semantics

Procedure Call (if $x_0 \cap FV_a\ q = \varnothing$ *and* $vs \cap FV_a\ q = \varnothing$*):*

$$\frac{\begin{array}{c} WF_c\ (\textbf{call}\ p(es))\ \rho, \quad WF_{env}\ \rho \\ \rho(p) = \langle vs, glbs, pre, post, c\rangle \\ x = vs\ \&\ glbs, \quad x_0 = logicals\ x, \quad x_0 \cap FV_a\ q = \varnothing, \quad vs \cap FV_a\ q = \varnothing \end{array}}{\{\ (pre \wedge ((\forall x.\ (post \Rightarrow q)) \lhd [x / x_0])) \lhd [vs := es]\ \}\ \textbf{call}\ p(es)\ \{q\}\ /\ \rho}$$

Procedure Call (if $vs \cap FV_a\ q = \varnothing$*):*

$$\frac{\begin{array}{c} WF_c\ (\textbf{call}\ p(es))\ \rho, \quad WF_{env}\ \rho \\ \rho(p) = \langle vs, glbs, pre, post, c\rangle \\ x = vs\ \&\ glbs, \quad x_0 = logicals\ x, \quad x'_0 = variants\ x_0\ (FV_a\ q), \quad vs \cap FV_a\ q = \varnothing \end{array}}{\{\ (pre \wedge ((\forall x.\ (post \lhd [x'_0 / x_0] \Rightarrow q)) \lhd [x / x'_0])) \lhd [vs := es]\ \}\ \textbf{call}\ p(es)\ \{q\}\ /\ \rho}$$

Procedure Call (general case):

$$\frac{\begin{array}{c} WF_c\ (\textbf{call}\ p(es))\ \rho, \quad WF_{env}\ \rho \\ \rho(p) = \langle vs, glbs, pre, post, c\rangle \\ x = vs\ \&\ glbs, \quad vs' = variants\ vs\ (FV_a\ q \cup glbs), \quad x' = vs'\ \&\ glbs \\ x_0 = logicals\ x, \quad x'_0 = variants\ (logicals\ x')\ (FV_a\ q) \end{array}}{\begin{array}{l}\{\ (pre \lhd [vs' / vs] \wedge ((\forall x'.\ (post \lhd [vs', x'_0 / vs, x_0] \Rightarrow q)) \lhd [x' / x'_0])) \lhd [vs' := es]\ \} \\ \qquad \textbf{call}\ p(es)\ \{q\}\ /\ \rho\end{array}}$$

Table 7. Programming Language Axiomatic Semantics (Continued)

The most interesting of these proofs, apart from the procedure call rules, was that of the **while**-loop rule. It was necessary to prove a subsidiary lemma first, by the strong version of rule induction for command semantics provided by Tom Melham's rule induction package. This lemma thus used versions of itself for "lower levels" in the relation built up by rule induction to prove each instance, and so needed strong induction to present as a usable assumption each hypothesized lower-level tuple in the relation. The subsidiary lemma was necessary because the **while**-loop rule as a theorem was not in the right syntactic form for the induction tactic. The lemma we proved is

$$\begin{array}{l}\vdash\ \forall a\ b\ c\ p\ q.\ \ \{p\}c\{a\}\ \ \wedge\ \ \{a \wedge (AB\ b)\}b\{p\}\ \ \wedge\ \ \{a \wedge \sim(AB\ b)\}b\{q\}\ \ \Rightarrow \\ \qquad \forall w\ s_1\ s_4.\ C\ w\ s_1\ s_4 \Rightarrow \\ \qquad\qquad ((w\ =\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ ) \Rightarrow (A\ a\ s_1 \Rightarrow A\ q\ s_4))\end{array}$$

The Rule of Adaptation enables one to take a previously proven partial correctness statement, $\{x_0 = x \wedge pre\}\ c\ \{post\}\ /\rho$, and derive an adaptation of this to a situation where c is being considered with a given postcondition q. The basic idea is given in the first version presented here; the general case handles situations where variables in q conflict with the x_0 variables that are in some sense "local" to the original partial correctness statement. This is actually a simpler version of the Rule of Adaptation than those previously presented in the literature; the simplicity arises from our dependence on the definition of proper substitution to automatically rename the bound variables x in the expression $\forall x.\ (post \Rightarrow q)$ under the substitution $[x\ /\ x_0]$. This renaming was performed manually in previous expressions of the Rule of Adaptation.

In a similar fashion, the three versions of the Procedure Call Rule range from the simplest, which shows the basic idea best, to the completely general, which has no restrictions on its applicability. The general case is the rule used in the VCG presented later. These were proven using the Rules of Adaptation above, combined with the definition of a well-formed environment. The environment contributed some of the necessary preconditions for using the Rules of Adaptation. By inspecting these rules, the reader will recognize a constructive method for creating an appropriate weakest precondition for a procedure call, given the postcondition. These Rules of Procedure Call were by far the most difficult theorems proven in this entire exercise, requiring very careful management of the variables involved in each subexpression and the precise meaning of the various substitutions. We believe that the subtleties involved here are so great as to constitute a compelling argument for mechanical proof-checking. Although one's intuition is useful and necessary, it is almost impossible to intuitively grasp all the issues involved.

Although we did prove analogous theorems as an axiomatic semantics for both the numeric and boolean expressions in the programming language, it turned out that there was a better way to handle them provided through the use of the translation functions. Using these translation functions, we may define functions to compute the appropriate precondition to an expression, given the postcondition, as follows.

`vexp`	ve_pre *e v* = *v* ◁ (*VE_state e*) vb_pre *b v* = *v* ◁ (*AB_state b*)
`aexp`	ae_pre *e a* = *a* ◁ (*VE_state e*) ab_pre *b a* = *a* ◁ (*AB_state b*)

Table 8. Expression Precondition Functions

We may now prove the following axiomatic semantics for expressions:

Numeric expression precondition:	*Boolean expression precondition:*
$\overline{\{\text{ae\_pre } e\ q\}\ e\ \{q\}}$	$\overline{\{\text{ab\_pre } b\ q\}\ b\ \{q\}}$

Table 9. Programming Language Expression Axiomatic Semantics

These precondition functions now allow us to revise the rules of inference for conditionals and loops, as follows.

Conditional:

$$\frac{\{r_1\}\ c_1\ \{q\}\ /\rho, \qquad \{r_2\}\ c_2\ \{q\}\ /\rho}{\{AB\ b\ \Rightarrow \text{ab\_pre } b\ r_1 \mid \text{ab\_pre } b\ r_2\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}\ /\rho}$$

Iteration:

$$\frac{\{a \wedge AB\ b \Rightarrow \text{ab\_pre } b\ p\} \quad \{a \wedge \sim(AB\ b) \Rightarrow \text{ab\_pre } b\ q\} \quad \{p\}\ c\ \{a\}\ /\rho}{\{a\}\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}\ /\rho}$$

Table 10. Programming Language Axiomatic Semantics (revisions)

11 Semantic Stages

We are almost ready to define and prove correct a VCG function for programs. Two correctness properties we wish to show are

vcgc_THM $\quad \vdash\ \forall c\ p\ q\ \rho.\ WF_{env}\ \rho\ \wedge\ WF_c\ c\ \rho\ \Rightarrow\ ALL\_EL$ **close** $(vcgc\ p\ c\ q\ \rho)\ \Rightarrow$ {p}c{q} / ρ

vcgd_THM $\quad \vdash\ \forall d\ \rho.\ (\rho = mkenv\ d\ \rho_0)\ \wedge\ WF_d\ d\ \rho\ \wedge\ ALL\_EL$ **close** $(vcgd\ d\ \rho)\ \Rightarrow\ WF_{env}\ \rho$

In order to prove vcgd_THM, we would like to use vcgc_THM, proven before. However, a problem arises. The vcgd_THM is used to prove the well-formedness of an environment. Given the syntactic well-formedness arising from $WF_d\ d\ \rho$ and the semantic well-formedness following from proving the verification conditions returned by *vcgd d* ρ, $WF_{env}\ \rho$ intuitively should follow. To reason from the verification conditions to the actual partial correctness of each procedure body, it is necessary to use vcgc_THM. The problem is that vcgc_THM itself requires a well-formed environment as a precondition! Thus it seems to be necessary to know that the environment is well-formed before we can prove that it is well-formed, a circular argument.

The solution is 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 will signify an ability to call procedures to one more level of calling depth. Thus, index 0 will designate 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 issue procedure calls, etc. In order to

define stages of well-formedness, we need to establish stages of command partial correctness specifications, and of the command semantic relation itself.

Without giving the full definition of the staged command semantic relation C_k, it suffices to say that it has one new argument k, which is the stage number, and that 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 tuple (regarding the procedure's body). We then define $\{p\}c\{q\}/\rho,k$ and $WF_{envk}\ \rho\ k$ similar to before but using the staged versions of the semantic predicates. Using these definitions, we can prove many staged versions of previous theorems, and also

$$\vdash\ \forall c\ \rho\ s_1\ s_2.\ \ C\ c\ \rho\ s_1\ s_2 = (\exists k.\ C_k\ c\ \rho\ k\ s_1\ s_2)$$

$$\vdash\ \forall p\ c\ q\ \rho.\ \ \{p\}c\{q\}/\rho = (\forall k.\ \{p\}c\{q\}/\rho,k)$$

$$\vdash\ \forall \rho.\ \ WF_{env}\ \rho = (\forall k.\ WF_{envk}\ \rho\ k)$$

This last theorem gives us the means to prove that an environment is well-formed. We first prove that for k=0, the antecedents of vcgd_THM imply the environment is well-formed to stage 0. Then, assuming those antecedents and that the environment is well-formed to stage k, we prove that it is well-formed to stage k+1. By induction, it is then well-formed for all stages, and by the above theorem, the environment is completely well-formed.

12 Verification Condition Generator

We now define a verification condition generator for this programming language. To begin, we first define a helper function *vcg1*, of type `cmd->aexp->env->(aexp # (aexp)list)`. This function takes a command a postcondition, and an 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 environment. This function does most of the work of calculating verification conditions.

The other verification condition generator functions, *vcgc* for commands, *vcgd* for declarations, and *vcg* for programs are defined similarly. Each returns a list of the verification conditions needed to verify the construct.

In these definitions, comma (,) makes a pair of two items, square brackets ([]) delimit lists, semicolon (;) within a list separates elements, and ampersand (&) is an infix version of HOL's `APPEND` operator to join two lists.

vcg1	*vcg1* (**skip**) $q\ \rho$ = q, [] *vcg1* (**abort**) $q\ \rho$ = **true**, [] *vcg1* (x := e) $q\ \rho$ = $q \lhd [x := e]$, [] *vcg1* (c_1 ; c_2) $q\ \rho$ = **let** (r,h_2) = *vcg1* $c_2\ q\ \rho$ **in** **let** (p,h_1) = *vcg1* $c_1\ r\ \rho$ **in** p, $(h_1$ & $h_2)$ *vcg1* (**if** b **then** c_1 **else** c_2) $q\ \rho$ = **let** (r_1,h_1) = *vcg1* $c_1\ q\ \rho$ **in** **let** (r_2,h_2) = *vcg1* $c_2\ q\ \rho$ **in** (*AB* b => ab_pre $b\ r_1$ | ab_pre $b\ r_2$), $(h_1$ & $h_2)$ *vcg1* (**assert** a **while** b **do** c) $q\ \rho$ = **let** (p,h) = *vcg1* $c\ a\ \rho$ **in** a, [$a \wedge AB\ b \Rightarrow$ ab_pre $b\ p$; $a \wedge \sim(AB\ b) \Rightarrow$ ab_pre $b\ q$] & h *vcg1* (**call** $p(es)$) $q\ \rho$ = **let** $(vs, glbs, pre, post, c) = \rho(p)$ **in** **let** $x = vs$ & $glbs$ **in** **let** $vs' = variants\ vs\ (FV_a\ q \cup glbs)$ **in** **let** $x' = vs'$ & $glbs$ **in** **let** $x_0 = logicals\ x$ **in** **let** $x'_0 = variants\ (logicals\ x')\ (FV_a\ q)$ **in** $((pre \lhd [vs' / vs]) \wedge ((\forall x'.\ (post \lhd [vs', x'_0 / vs, x_0] \Rightarrow q))$ $\lhd [x' / x'_0])) \lhd [vs' := es]$, []

Table 11. Verification Condition Generator

vcgc	*vcgc p c q ρ* = **let** (*r*,*h*) = *vcg1 c q ρ* **in** [*p* ⇒ *r*] & *h*
vcgd	*vcgd* (**proc** *p vs glbs pre post c*) *ρ* = **let** *x* = *vs* & *glbs* **in** **let** x_0 = *logicals x* **in** *vcgc* (x_0 = *x* ∧ *pre*) *c post ρ* *vcgd* (d_1; d_2) *ρ* = **let** h_1 = *vcgd* d_1 *ρ* **in** **let** h_2 = *vcgd* d_2 *ρ* **in** h_1 & h_2
vcg	*vcg* (**program** *d* ; *c* **end program**) *q* = **let** *ρ* = *mkenv d* $ρ_0$ **in** **let** h_1 = *vcgd d ρ* **in** **let** h_2 = *vcgc* **true** *c q ρ* **in** h_1 & h_2

Table 11. Verification Condition Generator (Continued)

The correctness of these VCG functions is established by proving the following theorems from the axioms and rules of inference of the axiomatic semantics:

vcg1_THM	⊢ ∀*c q ρ*. WF_{env} *ρ* ∧ WF_c *c ρ* ⇒ **let** (*p*,*h*) = *vcg1 c q ρ* **in** (*ALL_EL* **close** *h* ⇒ {*p*}*c*{*q*}/*ρ*)
vcgc_THM	⊢ ∀*c p q ρ*. WF_{env} *ρ* ∧ WF_c *c ρ* ⇒ *ALL_EL* **close** (*vcgc p c q ρ*) ⇒ {*p*}*c*{*q*}/*ρ*
vcgd_THM	⊢ ∀*d ρ*. *ρ* = *mkenv d* $ρ_0$ ∧ WF_d *d ρ* ∧ *ALL_EL* **close** (*vcgd d ρ*) ⇒ WF_{env} *ρ*
vcg_THM	⊢ ∀*π q*. WF_p *π* ∧ *ALL_EL* **close** (*vcg π q*) ⇒ *π*{*q*}

Table 12. Verification of Verification Condition Generator

ALL_EL P lst is defined in HOL as being true when for every element *x* in the list *lst*, the predicate *P* is true when applied to *x*. Accordingly, *ALL_EL* **close** *h* means that the universal closure of each verification condition in *h* is true.

These theorems are proven from the axiomatic semantics by induction on the structure of the construct involved. vcgd_THM relies on the induction by semantic stages discussed earlier, and enables the proof of vcg_THM. This verifies the VCG. It shows that the *vcg* function is *sound*, that the correctness of the verification conditions it produces suffice to establish the partial correctness of the annotated program. This does not show that the *vcg* function is *complete*, 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 [3]. However, this soundness result is quite useful, in that we may directly apply these theorems in order to prove individual programs partially correct within HOL, as seen in the next section.

13 Example Programs

Given the *vcg* function defined in the last section and its associated correctness theorem, proofs of program correctness may now be partially automated with security. This has been implemented in an HOL tactic, called `VCG_TAC`, which transforms 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.

We have proven the quotient/remainder algorithm, an example of two mutually recursive procedures to decide even/odd, and a procedural version of McCarthy's "91" function. As an example of the use of procedures, we will take a procedure to calculate triangle numbers. Triangle numbers are numbers in the series 1, 1+2, 1+2+3, …, which can be drawn as a triangle of dots, with one more dot in each row than in the row above. The program to be verified, with the annotations of the loop invariant and procedure pre- and postconditions, is established as the current goal by the following. Other than the introduction of fonts, bold-face and italics, here is exactly what is input to HOL:

```
g [[  program
          procedure triangle(val n);
               global a;
               pre  true;
               post  2 * a = ^n * (^n + 1);

               if n = 0 then a := 0
               else
                    triangle(n – 1);
                    a := a + n
               fi
          end procedure;

          triangle(4)

     end program
     { a = 10 }
  ]];;
```

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. The *triangle* procedure takes as input *n*, computes the *n*th triangle number and stores it in the global *a*. The specification of the procedure states that at the end of a call, *a* contains $n(n+1)/2$, which is a formula for the *n*th triangle number. The actual code of the procedure computes by recursive calls, reminiscent of counting the dots of the triangle, row by row.

Applying VCG_TAC to this goal produces the following two verification conditions:

VC1: $\forall n_1\ a.\ (2 * a = 4 * (4 + 1)) \Rightarrow (a = 10)$

VC2: $\forall \hat{}n\ n\ \hat{}a\ a.\ (\hat{}n = n) \wedge (\hat{}a = a) \Rightarrow$
$((n = 0) \Rightarrow (2 * 0 = \hat{}n * (\hat{}n + 1))) \wedge$
$(\sim(n = 0) \Rightarrow$
$(\forall n_2\ a'.\ (2 * a' = (n - 1) * ((n - 1) + 1)) \Rightarrow$
$(2 * (a' + n) = \hat{}n * (\hat{}n + 1))))$

Here is a transcript of the application of VCG_TAC to this problem. We have turned on the flag "print_vcg", which causes the tactic to print a trace of its processing of the program, in terms of the correctness relationships it forms at intermediate points. These are pretty-printed displays of the program and assertion language phrases constructed.

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

By the "ASSIGN" rule, we have
   [[ {2 * 0 = ^n * (^n + 1)} a := 0 {2 * a = ^n * (^n + 1)} ]]

By the "ASSIGN" rule, we have
   [[ {2 * (a + n) = ^n * (^n + 1)} a := a + n {2 * a = ^n * (^n + 1)} ]]

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

By the "SEQ" rule, we have
   [[ {true /\ !n2. !a. 2 * a = (n - 1) * ((n - 1) + 1) ==>
                        2 * (a + n) = ^n * (^n + 1)}
       triangle(n - 1); a := a + n {2 * a = ^n * (^n + 1)} ]]

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

```
By precondition strengthening, we have
   [[ {(^n = n /\ ^a = a /\ true) /\ true}
       if n = 0 then a := 0 else triangle(n - 1); a := a + n fi
      {2 * a = ^n * (^n + 1)} ]]
with additional verification condition
   [[ {(^n = n /\ ^a = a /\ true) /\ true ==>
       (n = 0 ==> 2 * 0 = ^n * (^n + 1)) /\
       (~(n = 0) ==>
        true /\ !n2. !a. 2 * a = (n - 1) * ((n - 1) + 1) ==>
                         2 * (a + n) = ^n * (^n + 1))} ]]

For the main body,

By the "CALL" rule, we have
   [[ {true /\ !n1. !a. 2 * a = 4 * (4 + 1) ==> a = 10} triangle(4)
      {a = 10} ]]

By precondition strengthening, we have
   [[ {true} triangle(4) {a = 10} ]]
with additional verification condition
   [[ {true ==> true /\ !n1. !a. 2 * a = 4 * (4 + 1) ==> a = 10} ]]

2 subgoals
"!n1 a. (2 * a = 4 * (4 + 1)) ==> (a = 10)"

"!^n n ^a a. (^n = n) /\ (^a = a) ==>
                ((n = 0) ==> (2 * 0 = ^n * (^n + 1))) /\
                (~(n = 0) ==>
                    (!n2 a'. (2 * a' = (n - 1) * ((n - 1) + 1)) ==>
                                (2 * (a' + n) = ^n * (^n + 1))))"

() : void
Run time: 507.3s
Garbage collection time: 46.0s
Intermediate theorems generated: 24269
```

These verification conditions are HOL Object Language conditions. The Object Language variables involved in these verification conditions are constructed to have names similar to the original program variable names; if there is a non-zero variant number, it is appended to the variable name. Thus, if one changed the name of program variable *n* to *k* in the example above, the verification conditions would be the same but with the OL variable *k* in place of *n*, k_1 in place of n_1, etc. The *a'* variables above were constructed by `ALPHA_CONV`, as part of the process of converting quantified assertion language variables into OL variables in `INTERPRET_aexp_CONV` in step (e) below.

As a second example, we consider McCarthy's "91" function. We define the function *f*91 as follows:

$$f91 = \lambda y.\ y > 100 \Rightarrow y - 10 \mid f91(f91(y + 11))$$

We claim that the behavior of *f*91 is such that

$$f91 = \lambda y.\ y > 100 \Rightarrow y - 10 \mid 91$$

which is not immediately obvious. Here is an expression of the "91" function as a goal for the VCG:

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

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

          skip
      end program
      {  true }
   ]];;
```

Here is a transcript of the application of VCG_TAC:

```
OK..
```

```
For procedure `p91`,

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

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

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

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

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

For the main body, *... elided ... here is the single verification condition produced:*

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

This VC is proven by taking four cases: $y < 90$, $90 \le y < 100$, $y = 100$, and $y > 100$.

Here is the HOL definition of the VCG_TAC tactic:

```
    let VCG_TAC =
(a)     MATCH_MP_TAC vcg_THM
(b)     THEN CONV_TAC (DEPTH_CONV WFp_CONV)
(c)     THEN CONV_TAC (DEPTH_CONV vcg_CONV)
(d)     THEN REWRITE_TAC[APPEND_INFIX;APPEND;ALL_EL;CLOSE]
(e)     THEN CONV_TAC (TOP_DEPTH_CONV INTERPRET_aexp_CONV)
(f)     THEN REWRITE_TAC[V_DEF]
        THEN CONV_TAC var_BND_CONV
        THEN REPEAT CONJ_TAC
        THEN ( GEN_TAC ORELSE ALL_TAC )
(g)     THEN INTERPRET_PROG_VARS_TAC;;
```

The VCG_TAC tactic first (a) applies the theorem VCG_THM, the last theorem of Table 12 of the previous section, to the current goal using the HOL tactic MATCH_MP_TAC to reason backwards from the program correctness statement to the invocation of the *vcg* function. By the theorem, the proof of these verification conditions will establish the proof of the original program correctness statement.

The next step of VCG_TAC is to "execute" the various syntactic manipulation functions mentioned in the current goal by symbolically rewriting the goal using the definitions of the functions. This applies (b) to the *WFp* well-formedness static check, and (c) to the *vcg* function itself. Because the rewriting process is done symbolically, instead of actually executing a program, it is relatively slow, but complete soundness is assured. This "execution" converts the invocation of the *vcg* function on the annotated program into the actual set of verification conditions that the *vcg* function returns.

The WFp_CONV used at (b) is built up from conversions which automatically check the well-formedness of declarations (WFd_CONV), commands (WFc_CONV), and individual variables (WFv_CONV), as previously defined.

The VCG_TAC tactic makes use at (c) of a set of conversions, culminating in vcg_CONV, to test the equality of variables (var_EQ_CONV), lookup a variable in a simultaneous substitution (var_BND_CONV), calculate a variant of a variable (variant_CONV), apply a substitution to an expression (subst_CONV), and reduce a term with calls to the *vcg* function in an efficient order (vcg_CONV), among others.

After performing these conversions, the program correctness goal is left as a set of "ground" verification conditions in the assertion language. VCG_TAC then (d–g) uses the definitions of the semantics of the assertion language to rewrite these verification conditions into equivalent statements in the Object Language of HOL, beginning with (d) the definition of **close**, then proceeding with (e) the definitions of *A* and (f) *V*. In particular, (e) all quantification over assertion language variables, and (g) all references to assertion language variables within program states, are converted to references to similarly-named OL variables. These verification conditions are then presented to the user as the necessary subgoals in the HOL Object Language that need to be solved in order to complete the proof of the program originally presented.

14 Future Work

In the future, we intend to extend this work to include several more language features, principally concurrency. In addition, we also intend to cover total correctness, beyond the partial correctness issues dealt with in this paper. We wish to find a method of proving the total correctness of systems of mutually recursive procedures, preventing infinite recursive descent, which is efficient and suitable for processing by a VCG.

Concurrency raises a whole 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.

15 Summary and Conclusions

The fundamental contribution of this work is the exhibition of a tool to ease the task of proving programs with mutually recursive procedures, which is itself proven to be sound. This verification condition generator tool performs an automatic, syntactic transformation of the annotated program into a set of verification conditions. The verification conditions produced are themselves proven within HOL, establishing the correctness of the program within the same system wherein the VCG was verified.

The relative complexity of the procedure call rule forms a compelling argument for the usefulness of machine-checked proof. There is not enough room in this paper to describe the intricate subtleties that arose in the proof. The history of unsound proposals for procedure calls indicate a need for stronger tools than intuition to construct such rules and verify their soundness.

This proof of the correctness of the VCG may be considered as an instance of a compiler correctness proof, with the VCG translating from annotated programs to lists of verification conditions. Each of these has its semantics defined, and the VCG correctness theorem closes the commutative diagram, showing that the truth of the verification conditions implies the truth of the annotated program.

The programming language and its associated assertion language are represented by new concrete recursive datatypes. This implies that they are completely independent of other data types and operations existing in the HOL system, without any hidden associations that might affect the validity of proof. This requires substantial work in defining their semantics and in proving the axioms and rules of inference of the axiomatic semantics from the operational semantics. However, this deeply embedded approach yields great expressiveness, ductility, and the ability to prove as theorems within HOL the correctness of various syntactic manipulations, which could only be stated as meta-theorems before. These theorems encapsulate a level of reasoning which now does not need to be repeated every time a program is verified, raising the level of proof from the semantic level to the syntactic. But the most important part of this work is the degree of trustworthiness of this syntactic reasoning. Verification condition generators are not new, but we are not aware of any other proofs of their correctness to this level of rigor. This enables program proofs which are both trustworthy and effective to a degree not previously seen together.

References

1. Sten Agerholm, "Mechanizing Program Verification in HOL", in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 208–222.

2. J. Camilleri and T. Melham, "Reasoning with Inductively Defined Relations in the HOL Theorem Prover", Technical Report No. 265, University of Cambridge Computer Laboratory, August 1992.

3. Stephen A. Cook, "Soundness and Completeness of an Axiom System for Program Verification", in *SIAM Journal on Computing*, Vol. 7, No. 1, February 1978, pp. 70–90.

4. G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wadsworth, *The ML Handbook* (INRIA, 1986).

5. Michael J. C. Gordon, "Mechanizing Programming Logics in Higher Order Logic", in *Current Trends in Hardware Verification and Automated Theorem Proving*, ed. P.A. Subrahmanyam and Graham Birtwistle, Springer-Verlag, New York, 1989, pp. 387–439.

6. Michael J. C. Gordon, and T. F. Melham, *Introduction to HOL, A theorem proving environment for higher order logic*, Cambridge University Press, Cambridge, 1993.

7. Peter V. Homeier and David F. Martin, "Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator", in *Proceedings of the 1994 International Workshop on the HOL Theorem Proving System and its Applications, Malta, September 1994*, Lecture Notes in C.S. Vol. 859, Springer-Verlag, pp. 269-284.

8. S. Igarashi, R. L. London, and D. C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation", *ACTA Informatica* 4, 1975, pp. 145–182.

9. Tom Melham, "A Package for Inductive Relation Definitions in HOL", in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 350–357.

10. L. C. Ragland, "A Verified Program Verifier", Technical Report No. 18, Department of Computer Sciences, University of Texas at Austin, May 1973.

11. Stefan Sokolowski, "Partial Correctness: The Term-Wise Approach", *Science of Computer Programming*, Vol. 4, 1984, pp. 141–157.

12. Allen Stoughton, "Substitution Revisited", *Theoretical Computer Science*, Vol. 59, 1988, pp. 317–325.

13. Glynn Winskel, *The Formal Semantics of Programming Languages, An Introduction*, The MIT Press, Cambridge, Massachusetts, 1993.