Mechanical Verification of Mutually Recursive Procedures

Peter V. Homeier and David F. Martin

UCLA Computer Science Department
Los Angeles, California 90024 USA
homeier@cs.ucla.edu and dmartin@cs.ucla.edu

Abstract. The verification of programs that contain mutually recursive procedures is a difficult task, and one which has not been satisfactorily addressed in the literature. Published proof rules have been later discovered to be unsound. Verification Condition Generator (VCG) tools have been effective in partially automating the verification of programs, but in the past these VCG tools have in general not themselves been proven, so any proof using and depending on these VCGs might not be sound. In this paper we present a set of proof rules for proving the partial correctness of programs with mutually recursive procedures, together with a VCG that automates the use of the proof rules in program correctness proofs. The soundness of the proof rules and the VCG itself have been mechanically proven within the Higher Order Logic theorem prover, with respect to the underlying structural operational semantics of the programming language. This proof of soundness then forms the core of an implementation of the VCG that significantly eases the verification of individual programs with complete security.

1 Introduction

Procedures appear in some form in nearly every programming language, because they extend the language with new phrases specific to the current problem. Parameters allow these new phrases to be used in a variety of contexts, and recursion allows procedures to be defined more simply. But with these capabilities come corresponding concerns for formally verifying these programs. First, the meaning of a procedure call depends on the definition of the procedure, remote from the call itself. Since the procedure is defined once and used many times, the definition should be verified once, and adapted for each instance of call. Recursion introduces issues of order, where a procedure must be verified before its body is. Finally, the passing of parameters has traditionally been a subject of great debate; investigation shows that this is a delicate and error-prone area. These qualities combine to make the task of verifying the partial correctness of a program with mutually recursive procedures arduous.

This difficulty may be ameliorated by 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 (VCs) 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 [5] and mechanically prove that the truth of the verification conditions it returns suffice to verify the partial correctness of 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 detail involved, relieving the programmer of all but the essential task of proving the verification conditions. This enables proofs of programs which are effective and trustworthy.

In a previous paper [8] we described such a VCG for a small **while**-loop programming language. The contribution of this paper is to extend the programming language considered to include mutually recursive procedures, including both variable and value parameters, and with access to global variables. We have also further extended this work to include termination and total correctness; this will be the subject of a forthcoming paper.

2 Previous Work

Several authors have treated recursive procedures, varying in the flexibility of the proof techniques and in the expressive power of the procedures themselves. The passing of parameters has been a delicate issue, with some proposals being later found unsound [7, 6]. This shows 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 to this, in that the security is maintained by the HOL system itself.

VCGs have been given in [12, 9, 11, 4, 8]. Of these, only Ragland's [12] and Homeier and Martin's [8] were verified. Gordon [4] did the original work of constructing within HOL a framework for proving the correctness of programs. 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 partial and total correctness as HOL tactics.

The shallow embedding approach yielded tools which could be used to soundly verify individual programs. However, the VCG tactics defined were not themselves proven. Instead, the resulting verification condition 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 [4].

This paper explores an 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.

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 a syntactic constructor of phrases in the programming language, and then separately define the semantics of each construct in a structural operational semantics [15]. This separation enables analyzing syntactic program phrases at the HOL Object Language level, and thus reasoning 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 is a recurring pattern in this work, where repeatedly a syntactic manipulation is defined, and then its semantics is described and proven correct in HOL.

Our previous paper [8] 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, involving both variable and value parameters, and allowing references to global variables. Many new concepts are introduced here. For example, programs must be checked for well-formedness before their execution or verification. This test needs to be performed only once, such as at compile time, as a static check. 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 to verify any individual program. Our approach has a special unity, as the proof rules, VCG, verification conditions, and individual programs are all proven correct within the Higher Order Logic mechanical theorem proving system in a connected fashion.

3 Higher Order Logic

Higher Order Logic (HOL) [5] 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") [3]; 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 create a deep embedding by defining a *third* level of language, contained within OL as concrete recursive datatypes, to represent the constructs of the programming language PL studied and its associated assertion language AL.

4 Programming and Assertion Languages

The syntax of the programming language PL is

$$
\begin{array}{rl}
\texttt{exp:}\ e ::= & n \mid x \mid {+}{+}\,x \mid e_1 + e_2 \mid e_1 - e_2 \\
\\
\texttt{bexp:}\ b ::= & e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \sim b \\
\\
\texttt{cmd:}\ c ::= & \textbf{skip} \mid \textbf{abort} \mid x := e \mid c_1 \,;\, c_2 \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \\
 & \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c \mid p(x_1, \ldots, x_n \,;\, e_1, \ldots, e_n) \\
\\
\texttt{decl:}\ d ::= & \textbf{procedure}\ p(\textbf{var}\ x_1, \ldots, x_n \,;\, \textbf{val}\ y_1, \ldots, y_n); \\
 & \quad \textbf{global}\ z_1, \ldots, z_n\,; \\
 & \quad \textbf{pre}\ a_1; \qquad\qquad \text{(this will be represented later as} \\
 & \quad \textbf{post}\ a_2; \qquad\qquad \textbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ c\text{)} \\
 & \quad c \\
 & \textbf{end procedure} \mid \\
 & d_1 \,;\, d_2 \\
\\
\texttt{prog:}\ \pi ::= & \textbf{program}\ d\,;\, c\ \textbf{end program}
\end{array}
$$

Table 1: Programming Language Syntax

Most of these constructs are standard. n is an unsigned integer (`num`); x and y are program variables, required not to begin with the character "^"; such 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 $p(xs; es)$, p is a string, xs is a list of variables, denoting the actual variable parameters (passed by call-by-name), and es is a list of `exp` expressions, denoting actual value parameters (call-by-value).

The procedure declaration specifies the procedure's name p, formal variable parameter names $x_1, \ldots, x_n$, formal value parameter names $y_1, \ldots, y_n$, global variables used in p (or any procedure p calls) $z_1, \ldots, z_n$, 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 declaration order. If two procedures are declared with the same name, the latter prevails. We will refer to a typical procedure declaration as **proc** *p vars vals glbs pre post c*, instead of the longer version given above.

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:`	$a ::= \textbf{true} \mid \textbf{false} \mid v_1 = v_2 \mid v_1 < v_2 \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid \sim a \mid$
	$a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid a_1$ `=>` a_2 `|` $a_3 \mid \textbf{close}\ a \mid \forall x.\, a \mid \exists x.\, a$

Table 2: Assertion Language Syntax

Most of these are standard. a_1 `=>` a_2 `|` 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. The constructor `AVAR` creates a `vexp` from a variable (`var`).

5 Operational Semantics

We define the type `state` as `var->num`, and the type `env` as
`string -> ((var)list # (var)list # (var)list # aexp # aexp # cmd)`,
representing an environment of procedure declarations, indexed by the name of the procedure. The tuple contains the variable parameter list, value parameter list, global variables list, the precondition, the postcondition, and the body.

The operational semantics of the programming language is expressed by

$E\ e\ s_1\ n\ s_2$:	numeric expression e`:exp` evaluated in state s_1 yields numeric value n`:num` and state s_2.
$B\ b\ s_1\ t\ s_2$:	boolean expression b`:bexp` evaluated in state s_1 yields truth value t`:bool` and state s_2.
$ES\ es\ s_1\ ns\ s_2$:	numeric expressions es`:(exp)list` evaluated in state s_1 yield numeric values ns`:(num)list` and state s_2.
$C\ c\ s_1\ s_2$:	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.

Table 3 gives the structural operational semantics [15] of the programming language PL, as rules inductively defining the six relations E, B, ES, C, D, and P. These relations (except for ES) are defined within HOL using Tom Melham's excellent rule induction package [1, 10]. First, we define some notation. We define ampersand (&) as an infix operator to append two lists. $[\,]$ is the empty list. The notation $f[e/x]$ indicates the function f updated so that

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

We will also use $f[es/xs]$ where es and xs are lists, to indicate a multiple update in order from right to left across the lists, so the right-most elements of es and xs make the first update, and the others are added on top of this.

variant $x\ s$ yields a variant of the variable x not in the set s; *variants* $xs\ s$ does the same for a list of variables xs.

For defining P, we define the empty environment $\rho_0 = \lambda p.\langle\, [\,],\ [\,],\ [\,],\ \mathbf{false},\ \mathbf{true},\ \mathbf{abort}\rangle$, and 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

$$\begin{aligned}
mkenv\ (\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ c)\ \rho &= \rho[\langle vars, vals, glbs, pre, post, c\rangle\ /\ p]\\
mkenv\ (d_1;\ d_2)\ \rho &= mkenv\ d_2\ (mkenv\ d_1\ \rho)
\end{aligned}$$

E	*Number* : $\dfrac{}{E\ (n)\ s\ n\ s}$	*Variable* : $\dfrac{}{E\ (x)\ s\ s(x)\ s}$	*Increment* : $\dfrac{E\ x\ s_1\ n\ s_2}{E\ (++x)\ s_1\ (n+1)\ s_2[(n+1)/x]}$
	Addition : $\dfrac{E\ e_1\ s_1\ n_1\ s_2,\quad E\ e_2\ s_2\ n_2\ s_3}{E\ (e_1 + e_2)\ s_1\ (n_1 + n_2)\ s_3}$	*Subtraction* : $\dfrac{E\ e_1\ s_1\ n_1\ s_2,\quad E\ e_2\ s_2\ n_2\ s_3}{E\ (e_1 - e_2)\ s_1\ (n_1 - n_2)\ s_3}$	
B	*Equality* : $\dfrac{E\ e_1\ s_1\ n_1\ s_2,\quad E\ e_2\ s_2\ n_2\ s_3}{B\ (e_1 = e_2)\ s_1\ (n_1 = n_2)\ s_3}$	*Less Than:* $\dfrac{E\ e_1\ s_1\ n_1\ s_2,\quad E\ e_2\ s_2\ n_2\ s_3}{E\ (e_1 < e_2)\ s_1\ (n_1 < n_2)\ s_3}$	
	Conjunction : $\dfrac{B\ b_1\ s_1\ t_1\ s_2,\quad B\ b_2\ s_2\ t_2\ s_3}{B\ (b_1 \wedge b_2)\ s_1\ (t_1 \wedge t_2)\ s_3}$	*Disjunction* : $\dfrac{B\ b_1\ s_1\ t_1\ s_2,\quad B\ b_2\ s_2\ t_2\ s_3}{B\ (b_1 \vee b_2)\ s_1\ (t_1 \vee t_2)\ s_3}$	*Negation* : $\dfrac{B\ b\ s_1\ t\ s_2}{B\ (\sim b)\ s_1\ (\sim t)\ s_2}$

Table 3: Programming Language Structural Operational Semantics

ES	*EmptyList* : $\dfrac{}{ES\ \mathbf{nil}\ s\ \mathbf{nil}\ s}$	*ConsList* : $\dfrac{E\ e\ s_1\ n\ s_2,\ \ ES\ es\ s_2\ ns\ s_3}{ES\ (\mathbf{cons}\ e\ es)\ s_1\ (\mathbf{cons}\ n\ ns)\ s_3}$
C	*Skip* : $\dfrac{}{C\ \mathbf{skip}\ \rho\ s\ s}$	*Conditional* : $\dfrac{B\ b\ s_1\ \mathrm{T}\ s_2,\ \ C\ c_1\ \rho\ s_2\ s_3}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ \rho\ s_1\ s_3}$
	Abort : (no rules)	$\dfrac{B\ b\ s_1\ \mathrm{F}\ s_2,\ \ C\ c_2\ \rho\ s_2\ s_3}{C\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ \rho\ s_1\ s_3}$
	Assignment : $\dfrac{E\ e\ s_1\ n\ s_2}{C\ (x := e)\ \rho\ s_1\ s_2[n/x]}$	*Iteration* : $\dfrac{\begin{array}{c} B\ b\ s_1\ \mathrm{T}\ s_2,\ \ C\ c\ \rho\ s_2\ s_3 \\ C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c)\ \rho\ s_3\ s_4 \end{array}}{C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c)\ \rho\ s_1\ s_4}$
	Sequence : $\dfrac{C\ c_1\ \rho\ s_1\ s_2,\ \ C\ c_2\ \rho\ s_2\ s_3}{C\ (c_1\ ;\ c_2)\ \rho\ s_1\ s_3}$	$\dfrac{B\ b\ s_1\ \mathrm{F}\ s_2}{C\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c)\ \rho\ s_1\ s_2}$
	Call : $\dfrac{\begin{array}{c} ES\ es\ s_1\ ns\ s_2 \\ \rho\ p = \langle vars, vals, glbs, pre, post, c\rangle \\ vals' = variants\ vals\ (xs\ \&\ glbs) \\ C\ (c \lhd [xs\ \&\ vals'\ /\ vars\ \&\ vals])\ \rho\ s_2[ns\ /\ vals']\ s_3 \end{array}}{C\ (\mathbf{call}\ p(xs;\ es))\ \rho\ s_1\ s_3[(\mathbf{map}\ s_2\ vals')\ /\ vals']}$	
D	*ProcedureDeclaration* : $\dfrac{}{\begin{array}{c} D\ (\mathbf{proc}\ p\ vars\ vals\ glbs\ pre\ post\ c)\ \rho \\ \rho[\langle vars, vals, glbs, pre, post, c\rangle\ /\ p] \end{array}}$	*Declaration Sequence:* $\dfrac{D\ d_1\ \rho_1\ \rho_2,\ \ D\ d_2\ \rho_2\ \rho_3}{D\ (d_1\ ;\ d_2)\ \rho_1\ \rho_3}$
P	*Program* : $\dfrac{D\ d\ \rho_0\ \rho_1,\qquad C\ c\ \rho_1\ s_0\ s_1}{P\ (\mathbf{program}\ d\ ;\ c\ \mathbf{end\ program})\ s_1}$	

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

	$V\ n\ s = n$
V	$V\ x\ s = s(x)$
	$V\ (v_1 + v_2)\ s = (V\ v_1\ s + V\ v_2\ s)$ $\quad$ ($-, *$ treated analogously)
	A **true** $s = \mathrm{T}$
	A **false** $s = \mathrm{F}$
	$A\ (v_1 = v_2)\ s = (V\ v_1\ s = V\ v_2\ s)$ $\quad$ ($<$ treated analogously)
A	$A\ (a_1 \wedge a_2)\ s = (A\ a_1\ s \wedge A\ a_2\ s)$
	($\vee, \sim, \Rightarrow, a_1 = a_2,\ a_1$ `=>` 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

6 Substitution and Translation

We present here a brief discussion of proper substitution and expression translation; for more details, see [8]. We define proper substitution on expressions using the technique of simultaneous substitutions, following Stoughton [14]. We represent substitutions by functions of type `subst` = `var->vexp`. This describes an infinite family of single substitutions, all of which are considered to take place simultaneously. The normal single substitution operation of $[v/x]$ may be defined as a special case:

$$[v/x] = \lambda y.\ (y = x \texttt{ => } v \texttt{ | AVAR } y).$$

and we also use the notation $[v_1, v_2, \ldots/x_1, x_2, \ldots]$ or $[vs/xs]$ for a multiple simultaneous substitution.

We apply a substitution by the infix operator $\triangleleft$. Thus, $a \triangleleft ss$ denotes the application of the simultaneous substitution ss to the expression a, where a can have type `vexp` or `aexp`. When a contains quantifiers, the $\triangleleft$ operation automatically induces a proper renaming of bound variables to avoid conflicts.

Expressions have typically not been treated in previous work on verification (except for [13]), and side effects have been particularly excluded. Consequently, expressions 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;

- a simultaneous substitution, representing the change in state from "before" evaluating e to "after" evaluating e.

The translator functions for numeric expressions are *VE* and *VE_state*, for lists are *VES* and *VES_state*, and for boolean expressions are *AB* and *AB_state*. 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:

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

7 Well-Formedness

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 must not appear within program code, but only within assertion language expressions.

A logical variable is designated by a special initial character, for which we use the caret (^) character. We reserve a space of names for logical variables by restricting program variables from beginning with this character. A state is a mapping from all variables, both logical and program variables, to their current integer values. We define a function *logical*:`var->var` to generate a logical variable from a program variable, by simply prefixing its name with a caret (^); we define a similar function, *logicals*, for lists of variables.

Well-formedness predicates test constructs to ensure that they are free from logical variables, and also perform other checks, such as ensuring that a procedure call has the right number of arguments for its definition. Well-formedness is defined by a predicate WF_φ for each kind of program phrase φ, where $\varphi \in \{s, x, xs, e, b, es, c, d, p\}$. We also define the predicate WF_{env} to test that an entire environment of procedures is correctly and consistently defined.

For the most part, program constructs are well-formed 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 calls must have no aliasing among the actual variable parameters and accessable globals;

4) procedure declarations must satisfy syntactic well-formedness conditions;

5) given syntactic well-formedness, environments must satisfy the partial correctness of each procedure body. These will be established later via a set of verification conditions and the axiomatic semantics.

8 Axiomatic Semantics

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

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

Table 5: Floyd/Hoare Partial Correctness Semantics

We now express the axiomatic semantics of the programming language in Table 6, where each rule is proven as a theorem from the structural operational semantics.

The most interesting of these proofs were those of the Rule of Adaptation and the Procedure Call Rule. The Rule of Adaptation enables one to take a previously proven partial correctness statement, $\{x0 = x \wedge pre\}\, c\, \{post\}/\rho$, and derive an adaptation of this to a situation where c is being considered with a given postcondition q. This is a simpler version of the Rule of Adaptation than those previously presented in the literature.

The Procedure Call Rule was proven using the Rule of Adaptation, combined with the definition of a well-formed environment. The environment contributed some of the necessary preconditions for using the Rule of Adaptation. By inspecting this rule, the reader will recognize a constructive method for creating an appropriate weakest precondition for a procedure call, given the postcondition. This Rule of Procedure Call was by far the most difficult theorem proven in this entire exercise. The effort of pushing the proof through HOL brought many subtle issues to light that had not been intuitively foreseen, and convinced us of the value of mechanically-checked proofs.

Skip :

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

Abort :

$$\frac{}{\{\mathbf{true}\}\ \mathbf{abort}\ \{q\}/\rho}$$

Assignment :

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

Sequence :

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

Conditional :

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

Iteration :

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

Rule of Adaptation :

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

Procedure Call :

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

Table 6: Programming Language Axiomatic Semantics

9 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 precondi-

	$\mathit{vcg1}\ (\mathbf{skip})\ q\ \rho = \quad q,\ [\,]$
	$\mathit{vcg1}\ (\mathbf{abort})\ q\ \rho = \mathbf{true},\ [\,]$
	$\mathit{vcg1}\ (x := e)\ q\ \rho = q \lhd [x := e],\ [\,]$
	$\mathit{vcg1}\ (c_1\ ;\ c_2)\ q\ \rho = \mathbf{let}\ (r, h_2) = \mathit{vcg1}\ c_2\ q\ \rho\ \mathbf{in}$
	$\mathbf{let}\ (p, h_1) = \mathit{vcg1}\ c_1\ r\ \rho\ \mathbf{in}$
	$p,\ (h_1\ \&\ h_2)$
	$\mathit{vcg1}\ (\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)\ q\ \rho =$
	$\mathbf{let}\ (r_1, h_1) = \mathit{vcg1}\ c_1\ q\ \rho\ \mathbf{in}$
	$\mathbf{let}\ (r_2, h_2) = \mathit{vcg1}\ c_2\ q\ \rho\ \mathbf{in}$
	$((AB\ b\ \Rightarrow\ \mathit{ab\_pre}\ b\ r_1)\ \wedge$
	$(\sim (AB\ b)\ \Rightarrow\ \mathit{ab\_pre}\ b\ r_2)),\ (h_1\ \&\ h_2)$
	$\mathit{vcg1}\ (\mathbf{assert}\ a\ \mathbf{while}\ b\ \mathbf{do}\ c)\ q\ \rho =$
	$\mathbf{let}\ (p, h) = \mathit{vcg1}\ c\ a\ \rho\ \mathbf{in}$
$\mathit{vcg1}$	$a, \quad [\ a \wedge AB\ b \Rightarrow \mathit{ab\_pre}\ b\ p\ ;$
	$a \wedge \sim (AB\ b) \Rightarrow \mathit{ab\_pre}\ b\ q\ ]\ \&\ h$
	$\mathit{vcg1}\ (\mathbf{call}\ p(xs; es))\ q\ \rho =$
	$\mathbf{let}\ (\mathit{vars}, \mathit{vals}, \mathit{glbs}, \mathit{pre}, \mathit{post}, c) = \rho\ p\ \mathbf{in}$
	$\mathbf{let}\ \mathit{vals}' = \mathit{variants}\ \mathit{vals}\ (FV_a\ q \cup (xs\ \&\ \mathit{glbs}))\ \mathbf{in}$
	$\mathbf{let}\ u = xs\ \&\ \mathit{vals}'\ \mathbf{in}$
	$\mathbf{let}\ v = \mathit{vars}\ \&\ \mathit{vals}\ \mathbf{in}$
	$\mathbf{let}\ x = u\ \&\ \mathit{glbs}\ \mathbf{in}$
	$\mathbf{let}\ y = v\ \&\ \mathit{glbs}\ \mathbf{in}$
	$\mathbf{let}\ x_0 = \mathit{logicals}\ x\ \mathbf{in}$
	$\mathbf{let}\ y_0 = \mathit{logicals}\ y\ \mathbf{in}$
	$\mathbf{let}\ x_0' = \mathit{variants}\ x_0\ (FV_a\ q)\ \mathbf{in}$
	$((\mathit{pre} \lhd [u\ /\ v]) \wedge ((\forall x.\ (\mathit{post} \lhd [u, x_0'\ /\ v, y_0] \Rightarrow q))$
	$\lhd [x\ /\ x_0'])) \lhd [\mathit{vals}' := es],\ [\,]$
vcgc	$\mathit{vcgc}\ p\ c\ q\ \rho = \quad \mathbf{let}\ (r, h) = \mathit{vcg1}\ c\ q\ \rho\ \mathbf{in}$
	$[p \Rightarrow r]\ \&\ h$
	$\mathit{vcgd}\ (\mathbf{proc}\ p\ \mathit{vars}\ \mathit{vals}\ \mathit{glbs}\ \mathit{pre}\ \mathit{post}\ c)\ \rho =$
	$\mathbf{let}\ x = \mathit{vars}\ \&\ \mathit{vals}\ \&\ \mathit{glbs}\ \mathbf{in}$
	$\mathbf{let}\ x_0 = \mathit{logicals}\ x\ \mathbf{in}$
vcgd	$\mathit{vcgc}\ (x_0 = x \wedge \mathit{pre})\ c\ \mathit{post}\ \rho$
	$\mathit{vcgd}\ (d_1\ ;\ d_2)\ \rho = \ \mathbf{let}\ h_1 = \mathit{vcgd}\ d_1\ \rho\ \mathbf{in}$
	$\mathbf{let}\ h_2 = \mathit{vcgd}\ d_2\ \rho\ \mathbf{in}$
	$h_1\ \&\ h_2$
	$\mathit{vcg}\ (\mathbf{program}\ d\ ;\ c\ \mathbf{end\ program})\ q =$
	$\mathbf{let}\ \rho = \mathit{mkenv}\ d\ \rho_0\ \mathbf{in}$
vcg	$\mathbf{let}\ h_1 = \mathit{vcgd}\ d\ \rho\ \mathbf{in}$
	$\mathbf{let}\ h_2 = \mathit{vcgc}\ \mathbf{true}\ c\ q\ \rho\ \mathbf{in}$
	$h_1\ \&\ h_2$

Table 7: Verification Condition Generator

tion, postcondition, and environment. This function does most of the work of calculating verification conditions. It uses the function *ab_pre b q*, which computes an appropriate precondition to the postcondition q, such that if *ab_pre b q* is true, then upon executing the programming language expression b, q must hold. For more details, please see [8].

The other verification condition generator functions, *vcgc* for commands, *vcgd* for declarations, and *vcg* for programs are defined with similar arguments. Each returns a list of the verification conditions needed to verify the construct. These verification condition generator functions are given in Table 7.

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	$\vdash \forall c\, q\, \rho.$	$WF_{env}\ \rho\ \wedge\ WF_c\ c\ \rho\ \Rightarrow$ $\mathbf{let}\ (p,h) = \mathit{vcg1}\ c\ q\ \rho\ \mathbf{in}$ $(\mathit{ALL\_EL}\ \mathbf{close}\ h \Rightarrow \{p\}\, c\, \{q\}/\rho)$
vcgc_THM	$\vdash \forall c\, p\, q\, \rho.$	$WF_{env}\ \rho\ \wedge\ WF_c\ c\ \rho\ \Rightarrow$ $\mathit{ALL\_EL}\ \mathbf{close}\ (\mathit{vcgc}\ p\ c\ q\ \rho) \Rightarrow \{p\}\, c\, \{q\}/\rho$
vcgd_THM	$\vdash \forall d\, \rho.$	$\rho = \mathit{mkenv}\ d\ \rho_0\ \wedge\ WF_d\ d\ \rho\ \wedge$ $\mathit{ALL\_EL}\ \mathbf{close}\ (\mathit{vcgd}\ d\ \rho) \Rightarrow WF_{env}\ \rho$
vcg_THM	$\vdash \forall \pi\ q.$	$WF_p\ \pi\ \wedge\ \mathit{ALL\_EL}\ \mathbf{close}\ (\mathit{vcg}\ \pi\ q) \Rightarrow \pi\, \{q\}$

ALL_EL P lst is defined in HOL as being true when for every element x in the list *lst*, 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 an additional induction by semantic stages, which resolves all issues of proving recursion for any individual program once at the meta-level.

This enables the proof of vcg_THM, which 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 *vcg* is *complete*, that if a program is correct, then *vcg* will produce a set of verification conditions sufficient to prove the program correct from the axiomatic semantics [2]. However, this soundness result is quite useful, in that we may directly apply this theorem in order to prove individual programs partially correct within HOL, as seen in the next section.

10 An Example Program

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 as 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.

As an example, we consider McCarthy's "91" function, defined as

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

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

$$f91 = \lambda y.\ y > 100 \texttt{ => } y - 10 \texttt{ | } 91$$

which is not immediately obvious. Here is the "91" function coded as a procedure:

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

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

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

The application of VCG_TAC to this goal yields the following VCs:

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

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

These verification conditions are HOL Object Language subgoals. The last VC is proven by taking four cases: $y < 90$, $90 \leq y < 100$, $y = 100$, and $y > 100$.

11 Summary and Conclusions

The fundamental contributions of this work are a system of proof rules and the corresponding VCG tool for the partial correctness of programs containing mutually recursive procedures. The soundness of these proof rules and VCG have been mechanically proven within the HOL theorem prover.

The relative complexity of the procedure call rule has convinced us of the usefulness of machine-checked proof. The history of unsound proposals indicates a need for stronger tools than intuition to verify such rules.

We have already found a method of proving the total correctness of systems of mutually recursive procedures, including termination, which is efficient and suitable for processing by a VCG, and have mechanically verified its soundness within HOL. We intend to extend this work to include several more language features, principally concurrency, which raises a whole host of new issues.

The most important result of this work is the degree of trustworthiness of the proof rules and the VCG tool. Verification condition generators are not new, but this level of rigor is. This enables program correctness proofs that are both effective and secure.

References

1. Camilleri, J., Melham, T.: Reasoning with Inductively Defined Relations in the HOL Theorem Prover. Technical Report No. 265, University of Cambridge Computer Laboratory, August 1992
2. Cook, S.: Soundness and Completeness of an Axiom System for Program Verification. SIAM Journal on Computing, Vol. 7, No. 1 (February 1978) 70–90
3. Cousineau, G., Gordon, M., Huet, G., Milner, R., Paulson, L., Wadsworth, C.: The ML Handbook. INRIA (1986)
4. Gordon, M.: Mechanizing Programming Logics in Higher Order Logic, in *Current Trends in Hardware Verification and Automated Theorem Proving*, ed. P.A. Subrahmanyam and G. Birtwistle. Springer-Verlag, New York (1989) 387–439
5. Gordon, M., Melham, T.: *Introduction to HOL, A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, Cambridge (1993)
6. Gries, D., Levin, G.: Assignment and Procedure Call Proof Rules. ACM TOPLAS **2** (1980) 564–579
7. Guttag, J., Horning, J., London, R.: A Proof Rule for Euclid Procedures, in *Formal Description of Programming Language Concepts*, ed. E.J. Neuhold, North-Holland, Amsterdam (1978) 211–220
8. Homeier, P., Martin, D.: A Mechanically Verified Verification Condition Generator. The Computer Journal **38** No. 2 (1995) 131–141
9. Igarashi, S., London, R., Luckham, D.: Automatic Program Verification I: A Logical Basis and its Implementation. ACTA Informatica **4** (1975) 145–182
10. Melham, T.: 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, ed. Archer, M., Joyce, J., Levitt, K., Windley, P. IEEE Computer Society Press (1992) 350–357
11. Moriconi, M., Schwartz, R.: Automatic Construction of Verification Condition Generators From Hoare Logics, in *Proceedings of ICALP 8*, Springer Lecture Notes in Computer Science 115 (1981) 363–377
12. Ragland, L.: A Verified Program Verifier. Technical Report No. 18, Department of Computer Sciences, University of Texas at Austin (May 1973)
13. Sokolowski, S.: Partial Correctness: The Term-Wise Approach. Science of Computer Programming **4** (1984) 141–157
14. Stoughton, A.: Substitution Revisited. Theoretical Computer Science **59** (1988) 317–325
15. Winskel, G.: *The Formal Semantics of Programming Languages, An Introduction.* The MIT Press, Cambridge, Massachusetts (1993)

This article was processed using the LaTeX macro package with LLNCS style