CHAPTER 3

Survey of Previous Research

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

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

3.1 Expressions with Side Effects

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

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

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

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

3.2 Procedures

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

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

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

Rule of Recursion:

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

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

Rule of Adaptation:

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

with additional restrictions, such as non-aliasing.

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

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

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

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

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

3.3 Total Correctness of Mutually Recursive Procedures

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

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

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

3.3.1 Sokołowski

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

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

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

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

This system did not deal with parameters.

3.3.2 Apt

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

3.3.3 America and de Boer

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

3.3.4 Pandya and Joseph

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

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

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

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

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

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

3.4.1 Ragland

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

3.4.2 Igarashi, London, and Luckham

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

3.4.3 Boyer and Moore

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

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

3.4.4 Gray

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

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

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

3.4.5 Gordon

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

3.4.6 Agerholm

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

3.4.7 Melham

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

3.4.8 Camilleri and Melham

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

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

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

3.4.10 Lin

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

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

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

3.4.11 Kaufmann

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

3.4.12 Homeier and Martin

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