CHAPTER 14

Future Research

"Thus says the LORD,

The Holy One of Israel, and his Maker:

'Ask Me of things to come concerning My sons;

And concerning the work of My hands, you command Me.' "

— Isaiah 45:11

"Whatever He hears He will speak; and He will show you things to come."

— John 16:13

In this chapter we consider possible future developments of the ideas presented in this work. These fall into four major areas: extensions to the programming and assertion languages, improvements to the VCG, implementations and tools to support the methodologies presented here, and proofs of completeness.

14.1 Language Extensions

There are many areas where we would like to extend the programming and assertion languages described here.

Probably the most immediate need is the inclusion of arrays. It is difficult to arrive a general, useful examples without arrays. This topic has been studied extensively before, so it should pose few theoretical difficulties. Some of the issues involved concern the inclusion of array bound checks in the preconditions computed by the VCG, the extension of the concept of aliasing to forbid confusion between array elements, and the passing of entire arrays as parameters.

The progress expressions currently permitted allow only the use of the operator $<$, implying the well-founded set of nonnegative integers. We expect to extend this to include the operator $\ll$, with the well-founded set of lists of nonnegative integers ordered lexicographically, and to include other well-founded sets and ordering relations. There does not appear to be any fundamental difficulty in adapting the proofs of recursiveness or termination to these additional forms. They would provide the ability to prove the termination of a wider variety of programs in ways that are natural and appropriate to the subjects of the programs.

In order to prove programs that implement certain recursive functions such as Ackerman's function, it will be necessary to extend the assertion language with user-defined functions, defined solely within the assertion language in order to abstract parts of the specifications. Even if no recursive functions are needed, such user-defined functions will be very practically useful in clearly expressing complex and layered specifications.

Many new operators can be added in a similar style to those already present. For example, if we add operators to perform integer division and check whether a number is odd or even, we can run Pandya and Joseph's example. In general, this seems to be one of the simplest and easiest extensions to accomplish, needing no theoretical additions. Nevertheless, we have not at this time expanded the language unnecessarily because of the great time and space issues that arise when defining new types in HOL which have many cases to represent the syntax trees.

One area of particular interest is the area of typing. A first extension would focus on adding valuable new base types, such as characters, strings, or bounded integers, for which there already exists support in the HOL logic. Further extensions could explore the creation of structured types such as records and arrays.

Input and output are important in bringing these systems closer to reality. We can model these as undetermined assignments to particular global variables, with assertions to act as preconditions restricting the possible input sequences. We would like to explore if the same translation techniques now used for the increment operator will also support input as an undetermined assignment.

One of the greatest challenges facing program verification is scaling up the theory to handle large, or even medium-sized programs, say of several tens of thousands of lines long. Possibly the only means will be through a form of modularization, where some program construct like Ada packages or Modula-2 modules will be used to encapsulate a section of the program with a well-defined interface. In the past these interfaces have incorporated only a syntactic specification, of the arity of each procedure and the types of its parameters. In the future we envision interfaces specifying the behavior and meaning of each module, just as preconditions and postconditions express that for procedures in this work. The point of the encapsulation is to modularize the proof of correctness of the program as well. Following the structure of the program, the proof should be structured so that each module can be independently verified apart from the rest of the program, perhaps with some required context as a precondition. Then the proofs of the verified modules should be adaptable for completing proofs of other parts of the program that use the modules. This situation is analogous on a larger scale to the specification and use of procedures in this work.

One of the most intriguing aspects of programming languages is nondeterminism, where either the order of subexpressions or the value of the operator itself may vary from one execution to the next. We would like to introduce an operator which nondeterministically selects an integer from 1 to n, so as to explore nondeterminism from the level of expressions up. Dijkstra's guarded conditional and repetition commands would be included as well. Nondeterminism may be handled by the same type of predicates for the operational semantics as are currently used; the final state will simply no longer be uniquely determined, but in fact these predicates will become true relations.

Finally, we hope someday to investigate the theoretically difficult area of concurrency. Concurrency raises a host of new issues, ranging from the level of structural operational semantics ("big-step" versus "small-step"), to dealing with assertions describing temporal sequences of states instead of single states, to issues of fairness. We believe that a proper treatment of concurrency will exhibit qualities of modularity and compositionality. *Modularity* means that a specification for a process should state both (a) the assumptions under which it should operate, and (b) the task (or commitment) which it should meet, given those assumptions. *Compositionality* means that the specification of a system of processes should be verifiable in terms of the specifications of the individual constituent processes.

14.2 VCG Improvements

We intend to continue to examine and improve the VCG functions for greater efficiency and ease of use, for example to reduce the number of verification conditions generated, especially those created through the analysis of the procedure call graph. One immediate improvement may be found by generating the verification conditions for each procedure in order. When the termination of a procedure was thus established, it would be deleted from the procedure call graph along with all incident arcs. This smaller call graph would then be the one used in generating verification conditions for the next procedure in order. Since there would be fewer arcs, there would be fewer cycles, and we anticipate far fewer verification conditions produced.

14.3 Implementations

We envision the theory developed in this work and others being supported by a variety of tools to ease the process of creating verified software. Proving programs correct is sufficiently difficult and full of details that mechanizing the task is a natural goal.

One tool would be a program editor, which would act as a structured editor for creating programs, but when a sufficiently substantial part was created (for example, a procedure) it would then automatically invoke the VCG on it. Then the verification conditions it produced would be collected and presented to the user to solve. The system could enforce the constraint that until all verification conditions were proven by the user, the code would not be submitted to the compiler, and thus could not be run.

In order to aid the user in proving these verification conditions, substantial theorem proving systems will have to be presented. We anticipate powerful graphical user interfaces to pictorially diagram the user's search for the correct proof. These would complement semi-automatic theorem provers running in the background, which would search for proofs of simple verification conditions or simple subgoals of a larger proof. This would eliminate the lower branches of the proof tree from the user's attention; and for most trees the lower branches contain the bulk of the tree's structure.

14.4 Completeness

Although we have not attempted any proof of completeness of this proof system, that does not mean that we think that unimportant. In the future we hope to create a proof of the system's *relative completeness*, in the sense of Cook [Coo78]. To some degree this will induce modifications of this approach, for completeness is a statement of what can be proven about a true program, and this would require encapsulating a proof system inside HOL.