CHAPTER 12

Significance

"LORD, make me to know my end,
And what is the measure of my days,
That I may know how frail I am."
— Psalm 39:4

In this chapter we will reflect and explore the significance of this work, and the possibility of its usefulness in the future.

The most novel part of this work is the development of a new methodology for proving the termination of programs with mutually recursive procedures. This includes new specifications to include in the headers of procedures, an algorithm for analyzing the procedure call graph to produce verification conditions, and logics for proving the termination of procedures from those verification conditions. We feel the approach is easier and simpler to use than previous proposals, while being more general in the sense of providing natural proofs of termination related to the program's original purpose. It also regularizes the proofs, making each example's proof less *ad hoc*, and structuring the proof according to the program logics. Furthermore, this methodology can be automated by a VCG, as we have done and exhibited in Chapter 8. This methodology should in general translate

to other programming languages, and we see this as a valuable technology for proving the termination of programs with procedures.

The most central thing we have learned from this work has been that the general approach we used was feasible. It was powerful, in that we could prove meta-theorems about all Sunrise programs, and it was effective, in that those proofs were accomplished once and would not need to be repeated for each application of the VCG. It was also quite difficult, in that there was considerable effort and skill required to accomplish the verification of the VCG.

In addition, the approach is quite solidly sound. Everything was established from the ground up, without claiming any new axioms, and only extending the theory by new definitions. Because we constructed a deep embedding of the programming and assertion languages within HOL, the types used to represent the abstract syntax trees were new types, without connections to or dependencies on previous parts of the theory. We established the semantics of the syntax trees ourselves by defining the operational semantics of the programming language and the denotational semantics of the assertion language. These semantics are simple and easily examined by the community, with their implications more easily understood than if we had taken an axiomatic semantics as the foundational definition. Then the axioms and rules of the axiomatic semantics were proven as theorems from the underlying foundational semantics, ensuring their soundness. Based on these sound axioms and rules, the VCG functions were verified and proven to be sound, which is our primary result.

The definitions and proofs are even more solidly secured by having created them within the HOL theorem proving environment, which ensures the soundness

of any theorems proven using its tools. For a user who is able to find the path to the goal of proving a theorem, HOL presents a powerful confirmation that that proof is in fact valid. HOL is generally understood to be weak in automating the search for a proof, say as compared with the Boyer-Moore theorem prover. Nevertheless, it was powerful and effective enough for our purposes here. Therefore, we can claim with assurance that this proof of soundness of this VCG has no logical errors. We have great confidence that the HOL-implemented proof is completely sound and trustworthy, and by extension, that the proofs of any programs proved using the VCG are likewise completely sound and trustworthy.

The idea of using a verification condition generator seems a useful and practical one, but this idea will need to be verified by actual experimentation and experience. The VCG we defined for total correctness seems quite satisfactory when it comes to the traditional analysis of the syntax of the program; there is room for improvement in the analysis of the call graph structure, as is discussed in Chapter 14.

The programming and assertion languages considered were quite small and not suitable for actual programming. This is because our goal was the exploration of the ideas behind certain program constructs, principally recursive procedures, and we included features that supported that goal. Nevertheless, it is not difficult to see how the languages could be extended with a more complete assortment of operators. This will be explored more in the next chapter.

The handling of expressions with side effects by the use of translation functions was elegant and surprisingly easy, once we had decided to use simultaneous substitutions to represent changes to the state. This part of the work has been

quite successful in handling our simple expressions. Future work will explore the applicability of this approach to more complex side effects.

The entrance and termination logics arose naturally during our work, and became the most convenient way to establish the verification of programs and the VCG itself. These are restricted versions of temporal logic, but powerful enough to accomplish the proofs of the recursiveness properties and the termination of procedures. It was important for us to develop some constraints on temporal logic, else it would not have been feasible to write a simple VCG to prove hypotheses written in such an expressive language.

Several realizations arose during the course of this work, and we present them here as understandings we have developed. These concern the separation of the programming and assertion languages, the need for well-formedness predicates, and the significant gap between partial and total correctness.

We believe that it is important to keep the ideas of the programming and assertion languages separate, and not confuse them, even if one's language does not include expressions with side effects. These two languages have different qualities and purposes, as was explored at the end of Section 5.5. One should not be beguiled by their overlap in appearance into assumming they are the same in essence.

Despite the relative lack of attention paid to date to well-formedness, we found this to be an area requiring a significant portion of the total effort. Perhaps the goal of complete formal verification of this system in every detail forced us to look at issues that previously were easy to dismiss. Just because an issue is obvious and part of common sense, does not mean that its formal verification

is inconsequential, either in effort required or in significance of the results. It appears to us that well-formedness will need to be a part of any practical VCG constructed in the future.

Finally, we feel that this work explores in a thorough way the difference between partial and total correctness of programs with mutually recursive procedures. The specifications required of the user for each procedure differed for specifying their partial correctness claims, using "**pre**" and "**post**", and their termination claims, using "**calls ... with**" and "**recurses with**". A respectable fraction of the total structure of the proof was principally concerned with proving total correctness; three out of the five program logics used were principally devoted to proving either termination or total correctness. Also, the structure of the proofs of partial and total correctness differed markedly. The proof of partial correctness worked by stages, proceeding by normal mathematical induction on the depth of recursive call to prove the entire environment well-formed for partial correctness. In contrast, the proof of total correctness involved an exploration of the procedure call graph to identify procedure call cycles and produce verification conditions which established the progress achieved around each cycle. Termination then followed based on a well-foundedness argument about infinitely decreasing sequences.

Clearly our tool would not be suitable for proving programs correct in an industrial setting. Rather, this has been a theoretical exploration of ideas in building a solid foundation for program proofs. In the future, these ideas may be of use to other researchers in building practical verification condition generators to help prove real programs.