L(T.PUSH(i).S)) & L(T.PUSH(i).TOP.S.TOP) → V(T.PUSH(i).TOP.S.TOP) = V(T.PUSH(i).S.TOP) .

The second problem is the more serious. If we ignore legality, then our method is limited in that we will be unable to prove some implementations correct that, in fact, are. For example, consider a program module that contains only the functions **PUSH** and **TOP**. If we can prove for a given implementation that **V(T.PUSH(i).TOP) = i** for all **T**, legal or not, then that implementation is correct. However, there will be correct implementations for which we will not be able to prove this. For example, we cannot prove **V(TOP.PUSH(i).TOP) = i** for any implementation that aborts when **TOP** is called on an empty stack. If we ignore equivalence, however, we are able to prove implementations correct that, in fact, are not. For example, if we ignore equivalence in our stack specification, an implementation that only handled **PUSH** and **TOP** would be provably correct.

Conclusion

Although employing implementation-oriented constructs such as hidden functions should be avoided, using traces to define procedure interaction leads to problems in doing correctness proofs. Nevertheless, these problems should be faced since current specification methods sacrifice specification abstractness for the ability to prove correctness. Work on the trace approach has the potential to yield a superior method of program specification and verification.

References

[1] McLean, J. A formal method for the abstract specification of software. *Journal of the ACM 31* (1984), 600-627.

Leonard G, Monk, MIT

OLD-FASHIONED LOGIC FOR VERIFICATION

As the title indirectly suggests, I believe we now need new logical foundations for verification technology. For example, instead of limiting our formal languages to talking about static (even if highly structured) specifications, and leaving the process of arriving at specifications to be guided by informal instructions for how to use the system, that process itself should be closely examined and crudely formalized, perhaps along the lines suggested in [1]. But that's all pie-in-the-sky. Here I shall outline four down-to-earth proposals for research, which require at most mild foundational extensions.

More importantly, these proposals are intended for what might be called "modular research." Results can hopefully be obtained by small projects and usefully transmitted to other independent workers. This is to avoid what seem to be two of our biggest obstacles: the significance of much work depends on the precise context of some larger project of which it is a part; and it is extremely difficult to coordinate and sustain the momentum of large projects. (I do not mean to imply that large, well funded, well motivated, and well directed projects are not ultimately necessary, just that we had better not depend exclusively on them.)

All suggestions are more or less original, but the first rather less than more. It concerns the hierarchical specification paradigm and was at least presaged by Dan Halpern. It has been explicitly mentioned in similar form by Richard Platek in a report to MITRE on the FDM methodology. The middle two ideas concern slight extension, with basically the same old tools, of what can be talked about in specifications. They have probably occurred to innumerable people. The last suggestion concerns automatic theorem proving and is being actively pursued this year at MITRE with internal research funding.

A. Which Interlevel Machines?

A critical variable in the general concept of hierarchical specification is the kind of connections between levels which are to be used. One alternative is to allow the use of a powerful language, even a real high order computer language, to describe how one level (thought of as an abstract machine) is implemented in terms of another. This allows very powerful control between levels. But it is hard to build a correct specification system with this approach, (much less to give a rigorous formalization), and difficult theorems are encountered when verifying even very

simple relations between levels. A conceptually clear alternative is to use "mappings" between levels in a way which amounts to logically defining the various concepts of one level in terms of those at the next level. It is easier to imagine a rigorous formal semantics being given for this approach, but there are indications that it is inconvenient to express as much interlevel control as one would like.

The suggestion is to write a careful specification of a realistic example on a toy specification system, using a small, non-deterministic programming language (but one which does have procedural constructs) to define abstract machines connecting the levels. Dijkstra's guarded command language (presented in [2]), might be appropriate for the interlevel specification. In any case the good common sense of the experimenter would be critical, because the complexity and degree of procedurality of the connecting machines should be kept as small as is consistent with achieving a comprehensible overall specification. Given this specification and a rigorous semantics for the toy system, one would generate appropriate theorems asserting, for instance, that the whole specification is a refinement of its top level. Proving these target theorems on some sort of toy theorem prover is definitely not intended, but they should be carefully judged informally. Approximately how unwieldy and difficult would the logically necessary theorems be in a typical example which is really done rigorously? We now have very little evidence to guess at an answer. Some would be provided by this experiment.

I imagine that the toy specification developed would lean towards the second alternative approach mentioned above. One important difference from some existing systems would be that state changes which are indecomposable at the top level could be decomposed in the finer language of a lower level into successive changes of its more refined version of the state. This should facilitate procedural control between levels. As regards the use of Dijkstra's guarded control language, it is worth noting that his view of what constitutes a legitimate non-deterministic machine does not include the extreme lack of determininsm which is not at all unusual in specifications we have grown used to, see [2] chapters 2 and especially 9. This could be interpreted as implicit sage advice to restrict the non-determinism, or at least as a warning about what a task that might be.

One application of this research would be to guide the development of future hierarchical specification systems. The experiment should at least provide a suggestion for a language for specifying interlevel control mechanisms and a bound on how elaborate those mechanisms should be. It would not be sensible to try to build a real system whose intended interlevel machines "fail" this pre-test, i.e., which result in unacceptably complex target theorems, at least if we expect ever to give it a rigorous semantic basis. Another application would be to automatic theorem proving for specification systems. It is to be expected that more ambitious systems will put greater demands on their theorem provers, but as of now it is hard for the ATP developers to get a realistic idea of what kind of target theorems they should aim to be able to handle.

B. Introducing the User to Reality

Part of the burden of a specification system should be to help make clear to the user some specific information about the difference between what the implementation will really do and the user's abstract picture of what it should do (as if things weren't bad enough already!). In a simple case this might amount to pointing out that it will really only take integer square roots of numbers less than some N. Unfortunately, the accommodations to reality one must make are much more complex than this, even a vocabulary to describe them is often unavailable.

The second project is to suggest a way of modifying existing systems to incorporate a crude ability to accomplish two goals: first, to agree with the user on an appropriate way of talking about "limitations of reality" on the system, such as isolating critical variables which must remain below certain bounds; second, to give very rough estimates on how large critical variables can be. One kind of rough estimate which it might be possible to give on the maximum M of x would be in terms of log log M; for instance, is the log base 2 of the log base 10 of M about 1, 2, 3, 4, or >5. Of course, a third possible goal is to assist in maximizing these bounds.

C. Counting Steps

The Scott-Strachey approach to denotational semantics for programming languages does not discriminate between two algorithms which compute the same function but which require wildly different amounts of various resources, in particular time, see [3] chapter 2. It may be beyond the capability of current specification systems to say much about resource usage, but we do want to do a little of this eventually, and that would be literally impossible if a formal semantics used such

an approach. There is a considerable body of work already done on this approach to semantics, and it is tempting to try using or imitating it for real specification systems.

This is an unfortunate situation, but I speculate that the developers of this approach could have included an ability to talk about time if they had considered it important enough. Their natural first stage was to do things as slickly as possible, and the importance of counting steps is only gradually becoming apparent, (witness the recent popularity of the phrase "combinatorial explosion"). This suggested project is to rework the Scott-Strachey approach to incorporate an ability to refer to the system clock. If the same model of the lambda calculus is to be used, the critical difference might be made by introducing an extra argument and output, time, to all functions. It may be ugly the first time through.

D. Heuristics for Theorem Provers

Attempting to avoid the black hole of infinite or effectively infinite computation requires all sophisticated automatic theorem provers to rely on systematic guessing procedures to search more quickly for proofs. These vary from simple tricks to accumulations of plausibly useful rules whose combined effect is unpredictable, see [4]. Improving the quality of these heuristics could be a big help in eventually building better theorem provers.

One difficulty is that heuristics are hard to pick out, they are often embodied implicitly in the programming of the theorem prover, not explicitly called as identifiable subroutines. Another problem is that there is no good way of comparing heuristics. Comparing the relative performance of their resident provers is far too indirect, and we don't have good ways of comparing theorem provers anyway.

The suggested research approach is to individuate heuristics as strategies for semantical debates. In this context a typical action of a "heuristic" would be to decide which of two assertions it believes is more likely to be true. It is convenient in the implementation actually being used to give them considerably more structure, but this is their paradigmatic mode of action. The semantical debates being used are an extension of an old reformulation of the notion of truth of mathematical assertions in terms of the existence of winning strategies.

The method of evaluation is to test the relative performance of heuristics in fair debates, which invloves a symmetric pairing of asserting true and false propositions. It is perfectly possible to win a debate as the asserter of a false proposition. In fact, the more difficult the proposition being debated, the more the outcome of the debate depends on the skill of the debaters, and the less it depends on the actual truth of the proposition. (Compare this situation to testing theorem prover directly to see which can prove things better. If the target theorems are too easy, then the results are meaningless, if they are interesting, and hence hard, then the score is likely to be 0 - 0.) Another method of evaluation is available, namely testing ones subjective impression of the quality of a heuristic by taking the other side against it personally. This provides a useful if informal connection to the outside world.

This project should be viewed as laying groundwork for a future theorem prover, rather than as a direct effort to build one. In its initial stage formal games with enough expressive power to talk about an interesting fragment of mathematics are being implemented, in a way which hopefully allows for quite a wide variety of heuristics to be tested.

References

[1] Scherlis, William, and Scott, Dana, "First Steps Towards Inferential Programming," CMU-CS-83-142, Dept. of Comp. Sci., Carnegie-Mellon University.

[2] Dijkstra, Edsger, "A Discipline of Programming," Englewood Cliffs, N.J.: Prentice-Hall 1976.

[3] Stoy, Joseph, "Denotational Semantics," Cambridge, Mass.: The MIT Press 1981.

[4] Boyer, Robert, and Moore, J Strother, "A Computational Logic," New York, N.Y.: Academic Press 1979.