A LOGICIAN LOOKS AT PROGRAMMING

by
Harvey M. Friedman
http://www.math.ohio-state.edu/~friedman/
May 31, 2000

I would like to barge in, uninvited, with full naivete, to design a programming environment, aimed especially at the general mathematical community.

Before continuing, let me apologize in advance to the distinguished software engineers who may have been curious enough to be here.

I will look forward to your comments before I try to publish in this area. Or alternatively, you and your colleagues in software engineering may in fact stop me from publishing in this area! But I think that there are some merits in having someone in mathematical logic and the foundations of mathematics tackle crucial issues in software engineering from their own perspective, head on, with no fear, discarding all of the talk about their being no magic bullets, etcetera.

I had some experience designing music performance software, and working with programmers. I was convinced that my specs had a rigor of their own, and certainly small changes could be made very easily in my specs. However, the implementation bills were not small. My specs just got translated into C and there was grossly inadequate communication. Nevertheless, the products got done and sold in small numbers. There were marketing problems, and so I proved that it is possible to lose serious money in high tech.

I have been talking to the software engineering group at Ohio State University, led by Bruce Weide and Bill Ogden. They don't think I am crazy, and in fact keep inviting me back to talk to them and their students.

1. Design a programming environement especially friendly for the general mathematical community.
2. Maximally leverage the thought processes of mathematicians.
3. Present environments are not so friendly in two ways.
4. First there is a lot of computerese. This is perhaps not too serious and special sugar for mathematicians should cure this.

5. More fundamentally, there are stages of computation which do not correspond cleanly to standard mathematical objects.
6. And there is the responsibility of memory management in many languages.
7. The closer programming is to the way mathematicians construct standard mathematical objects the easier it is to specificy and verify programs, either informally, semiformally, or formally.
8. Let us concentrate on basic programming of this type. One is programming a mathematical function which takes as input objects that are to be stored in the computer, and outputs objects to be stored in the computer. There is no interaction with the outside world.
9. The most mathematically friendly setup already goes back to Kleene's work on primitive recursive functions. Of course, implementation is completely impractical, and does not support data structures except natural numbers.
10. In the most friendly style of programming, the code consists of successive introduction of functions by explicit definition, each one defined explicitly in terms of previous ones. Go back to some primitives.
11. This is an extremely modular setup which supports verification at the informal, semiformal, and formal levels.
12. Of course, informal is not all that reliable, semiformal more reliable but time consuming, and formal extremely time consuming.
13. However, this reduces it to more or less mathematics, where projects like Mizar are busy at work. There are reasons to be somewhat optimistic about verifiers for math, even though it remains expensive.
14. But they are trying to verify serious math. Whereas the overwhelming majority of general purpose programming leads only to mathematical trivialities.
15. In fact, special software that is very good at the commonly occurring mathematical trivialities seems like a reasonable goal.
16. Want to unleash the great untapped power of the general mathematical community.
17. The programming style that is closest to what we have in mind is functional programming.

18. However, most people other than functional programming people think that it is unacceptably inefficient.
19. And that in order to fix the efficiency problems, one has to add control statements for greater control of computation, thereby defeating the purpose of functional programming.
20. These inefficiencies are connected with the usual implementation of functional programming, where copying of objects is necessary whenever those objects are the result of subroutines even if such objects are created from objects that are only incrementally different.
21. In imperative languages, one takes advantage of sharing of common locations when storing multiple objects. You directly control the form of storage and the location of storage.
22. We are optimistic about solving the efficiency problem for functional programming.
23. Since we are certainly not going to add control statements, we need to have provisions for the sharing of locations in incrementally different objects something that is supported in the primitives.
24. Thus we must set things up so that this sharing is propogated upward properly in a complex program.
25. We accomplish this through an appropriate choice of data structures at two layers: the semantic layer where the mathematician is most comfortable, and the implementational level where we view the machine is manipulating implementations of the semantic objects.
26. This requires the development of an implementational calculus that gives upper estimates on the resources used in the implementation as a function of the inputs.
27. This plan requires a careful inductive argument on the structure of a functional program.
28. Furthermore, it is important that in order to keep such an analysis honest, one uses a finite RAM model with reasonable word length and number of words of memory.
29. We then plan to write an informal compilier from functional programs into
finite RAM machine.

30. We then have to prove that the actual resources used are within the limits set by the implementational calculus.
31. Fast automatic memory management is used, which avoids time consuming garbage collection.
32. We must also import externally written code; i.e., we need to support relativized programming.
33. For this we must be careful to always make the distinction between functions to be implemented versus objects to be held at once in the computer.
34. We must also support processing data coming in from the outside world which may be time critical.
35. For the mathematical objects visible to the mathematician, we use hypersequences. At the bottom there are integers with exact arithmetic, and then we close off hierarchically with finite sequences of.
36. At the lower implementational level, we use a 2-3 tree representation of hypersequences, a convenient balanced tree scheme.