Is Scheme Faster than C?

Jonathan Sobel
Indiana University

February, 1994

During my first term of graduate school at Indiana University, I had the good fortune to be taking the courses Analysis of Algorithms and Programming Languages at the same time. One of the assignments in the Algorithms course was a major term project in which we were to implement and optimize the Fast Multiplication algorithm. Fast Multiplication is a recursive, divide-and-conquer algorithm for multiplying two numbers, especially large numbers: hundreds or thousands of digits. (It is also used at the hardware level, where the units are bits, instead of digits.)

Part of our grade for this project was based on how fast the program ran. Everyone else immediately started writing in C, and even some in Assembly. Of course, they all spent hours upon hours debugging as they tried to get their highly optimized programs running. Modify, re-compile, test; modify, re-compile, test; on and on....

To everyone's amazement and surprise, I started writing in Scheme. I had only seen Scheme for the first time two months before, in my Programming Languages course, but its simplicity made it attractive. Even more importantly, I had discovered the joy of incremental compilation: make a change or an addition without recompiling everything. I wonder how many hours I saved.... On the other hand, the speed of the program was the most important factor, and even though Chez Scheme (the Scheme implementation we use here at Indiana University) is amazingly fast, it can't quite keep up with C in most cases. So why choose Scheme?

Each call to **fast-multiply** produces three recursive calls to **fast-multiply** (or to a simple **multiply** routine, once you reach small enough numbers). What I noticed was that each of those three recursive calls had nearly the

same control context. In a simple-minded implementation in C, that context would be saved and restored three times, being destroyed after the return of each call. I thought to myself: If only I had explicit control over the flow of my program, I could speed it up significantly by creating that context only once and using it three times, destroying it only after the return of the third recursive call. Function calls are so costly! But I would never want to attempt such a thing from scratch in C.

What I had been learning in my Programming Language course, however, was that I really could manage my own control flow if I wanted. Furthermore, I could start with a simpler, more naive program and basically derive the sophisticated one by a series of correctness-preserving program transformations. This is where Scheme really won. Because of its extremely algorithmic—almost mathematical—nature, Scheme can be easily manipulated in a sort of algebraic style. One can follow a series of rewrite rules (just about blindly) to transform a program into another form with some desirable property. This was exactly what I needed.

I started by implementing the algorithm very directly, following the steps given in our Analysis of Algorithms textbook line by line. (I definitely did NOT spend much time debugging in this phase.) Then I converted the program to continuation passing style. Next I made the continuations into explicit record structures, rather than Scheme procedures. Finally, I transformed all the procedures to pass information via registers, instead of calling each other with arguments. At this stage, a procedure call is merely a simple **goto**, quite a cheap operation. I completed the entire transformation in a few hours.

With the Scheme program in its final form, it was a simple matter to translate it into C. The C program contained no function calls whatsoever; I really did use **goto**. I had never had the courage to use a **goto** in C before; it was just too dangerous a tool. It was kind of fun to use them now, knowing that I was completely safe in doing so. That was the amazing part: I had produced a program that I could not have written, and in any case would not have wanted to write.

Of course, you might be asking yourself, as I was, "But does it run fast?" Speed was, after all, my main goal. The answer is a very resounding "Yes!" Out of a class of about 15 students, only one person beat me (and only barely), and he wrote significant portions of his program directly in Assembly Language. The next fastest after me took nearly twice as long to do the same

work.

Now I know of several more transformations which I could have applied to my Scheme program before I translated it into C, which would have put mine in first place. A runtime profile of my program revealed that the majority of time was spent in the C routines **malloc** and **free**. I could have eliminated that heap usage by transforming my program into a form in which all data allocation and control management would have been completely stack-based, with an explicitly managed stack. I could have pushed onto the stack exactly that control information which I deemed necessary. There were places where I could have modified the existing stack record, rather than popping it off and creating a (similar) new one in its place. These transformations were also presented in my Programming Languages course. Unfortunately, at the time that I did my Algorithms project, I did not yet grasp them well enough to use them.

What I learned from this experience was the importance of a structured, systematic approach to programming. I have found that is it better to write a simple, direct program to solve a problem, having become convinced through experience that radical structural transformations can come later; and I can depend on those transformations to preserve the semantics of my program. I have also learned that I can be free to focus on the nature of whatever problem I am trying to solve, rather than on how efficient my solution is. Efficiency comes from elegant solutions, not optimized programs. Optimization is just a few correctness-preserving transformations away.