Statement of Purpose

Marie-Christine R. Chevalier

I fell in love with programming on the second day of my first computer science class. I remember the moment distinctly. The professor was demonstrating a program to draw a circle on the screen; she showed us the Pascal source code and then ran it. I was amazed that such a few simple lines of code could cause a complex machine to perform a complex action, and for the duration of the course I was able to think about little else.

Five years later, I am still fascinated by the relationship between simple programs and complex machines. In particular, the concept of levels of abstraction is one of the most exciting and beautiful ideas I have ever encountered. Abstraction is what makes it possible to write simple programs that correspond to arbitrarily complex sequences of machine instructions; now that I understand what was behind the program that so amazed me, I am no less amazed. Now, though, I see that programming languages have the potential to provide far higher levels of abstraction than the level provided by languages like Pascal. All higher-level languages put distance between the programmer and the bare machine, but imperative languages still encourage programmers to bind their thinking to the way the computer actually works – to think like a machine. Consciously thinking like a machine is not the most natural way for people to solve certain problems, which is why imperative programs are hard to debug and reason about in the large. Functional languages are different: they encourage a modular programming style which resembles mathematics more than machine instructions. Yet precisely for this reason, functional programs tend to be less efficient than their imperative counterparts. I am interested in studying ways to unleash the true power of functional languages by removing the efficiency constraints which have so far shackled them.

For my senior thesis, I am currently working on a project motivated by reducing this tradeoff between modularity and efficiency. A modular program creates many intermediate data structures, which exist solely by virtue of the program's modular structure and are not inherently necessary to perform the program's computation (we call these *virtual* data structures). Yet modularity is a mechanism for helping programmers write programs; it doesn't need to be preserved at the point when a machine actually runs the program, any more than a compiler needs to preserve comments in program text. This insight sparked the development of deforestation, a program transformation which removes virtual data structures from programs. Most implementations of deforestation which have been proposed to date rely on the input program being in a special, restricted form in order to work, but a new method for deforestation, type-inference-based deforestation, does not. Where methods such as short cut deforestation[1] rely on the input program using a special form called `foldr` to define functions which produce data structures and another form called `build` to define functions which consume them, the type inference algorithm only requires that `foldr` be used, and automatically derives `build` forms based on type information.[2] In theory, type-inference-based deforestation seems simpler and more flexible than previous methods. This year I am investigating whether or not this is really true, by fully implementing the type-inference-based deforestation algorithm.

I have gained research experience on two occasions. First, during summer 1999, I participated in the SUPERB program at Berkeley, and worked with Sanjoy Dasgupta (then a graduate student at Berkeley) and Professor Umesh Vazirani. My goal was to implement an algorithm for learning

[1] Andrew Gill, John Launchbury, and Simon Peyton Jones, "A Short Cut to Deforestation", Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93), pp. 223-232, 1993.

[2] Olaf Chitil, "Type-Inference Based Deforestation of Functional Programs", unpublished Ph.D thesis, 2000.

Gaussian mixtures, which was developed by Dasgupta. Dasgupta theorized that this algorithm was simpler than but more effective than the established algorithm for this problem. Working independently, I implemented the algorithm in C and experimented with running it on sample data sets; by doing so, I determined that the algorithm's performance could be improved by using random sampling to choose the initial guesses for means, rather than iterating over the entire data set. Second, I began my thesis work this summer as part of the Bates/Wellesley Lumberjack Project, which is ongoing and incorporates several students from Wellesley and Bates in research to compare different methods for deforestation. During the summer I familiarized myself with the basics of deforestation and with the Haskell language, and then chose type-inference-based deforestation as a project to research. I began the first phase of the project, extending Chitil's prototype in order to accept programs in Core (the intermediate language of the Glasgow Haskell Compiler), at the end of the summer, and am currently completing that phase.

I hope to pursue an academic career, as I have enjoyed these research experiences and I would like to remain in an environment where I can choose problems to research based on their intellectual, rather than commercial value. In addition, I have gained teaching experience by tutoring students in intermediate-level computer science classes at Wellesley for the past two years. I enjoy teaching both for the excitement of helping someone else understand an idea, and for the improved understanding it gives me of the subject at hand. I look forward to making a contribution to the study of programming languages, and learning everything possible about computer science as a whole, in graduate school.