Proposed Plan of Research

Marie-Christine R. Chevalier

Introduction

Functional programming can improve the reliability and security of software systems in unique ways, but many programmers reject it, arguing that the inefficiency associated with functional programming outweighs any of its possible advantages. I hope to strengthen the arguments in favor of functional programming by studying ways to reduce this inefficiency and to apply it to practical problems.

Programming shares features with both engineering and mathematics, but people involved in it are generally either practicing programmers who ignore the mathematics or theorists who ignore the engineering. This is unfortunate for programmers, because functional programming languages are better at promoting abstraction and modularity, and because many useful techniques – such as applying formal methods to programming – are unique to and particularly suited for functional languages [2]. It is also unfortunate for theorists of functional programming, whose ideas are being ignored in part due to their own lack of concern for efficiency. Because I am interested in both the theoretical aspects and the practical aspects of programming, I would like to do research which uses established theoretical tools to apply functional programming in new realms such as security. I am also interested in increasing the efficiency of functional languages' implementations, and am currently doing research which addresses that problem. The suitability of the functional paradigm for programming as an abstract intellectual activity is already well-demonstrated, and now is the time to apply it to programming as a concrete problem-solving tool as well.

Because I do not yet know where I will be attending graduate school, I cannot name a specific problem I will research. Instead, I will detail two examples of the kinds of problems that interest me.

Modularity Versus Efficiency

Every programming language which has achieved widespread popularity outside academic realms has been primarily based on the imperative paradigm. One major reason for this is efficiency: functional programs tend to require more resources to compile and run than imperative programs. In functional languages, programs are written in a highly modular style, with many small functions that interact with each other in complex ways. Achieving modularity often entails creating intermediate data structures, which require time to create and memory to store; hence, modularity is one reason why functional programs can be less than efficient.

Programmers are currently responsible for choosing the right balance between modularity and efficiency in their programs; functional programmers lean more towards modularity, and imperative programmers lean towards efficiency. 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. Methods such as deforestation[1] can remove some of the inefficiency associated with modularity, while retaining modularity's benefits to the programmer.

Security and Static Checking

Computer security is a new incarnation of an old problem: the trade-off between abstraction and flexibility. Security problems correspond to poorly designed abstraction barriers. Abstraction barri-

[1] described in my Previous Research Experience statement

ers, after all, are supposed to provide users with exactly as much functionality as they need. A good abstraction must not make unnecessarily confusing or potentially harmful procedure calls available to the user, but it also must not hide so much detail from the user as to make it impossible to use the abstraction.

Consider the problem of executable e-mail attachments. A typical PC e-mail client will allow the user to run an executable file received as an e-mail attachment, and may even include scripting features that allow senders to embed potentially harmful instructions in an e-mail that will be automatically executed by the recipient's e-mail client, without the recipient taking any action beyond opening the e-mail. The popularity of scripting and attachments has weakened the abstraction barrier between the sender of an e-mail and its recipient, a barrier that was stronger in the days when e-mail was restricted to plain text. The advent of viruses that spread via e-mail shows that the abstraction barrier is now *too* weak. An executable received via e-mail is probably not trustworthy enough to run in an environment where it may have access to any operation on the user's machine, yet an e-mail client shouldn't forbid running executable attachments altogether.

How would one design a better abstraction barrier? Virus protection software is not an adequate solution, as current virus protection software is only capable of searching a program for code that matches a known virus. Currently, the technology for determining (or at least giving a good approximation as to) whether an arbitrary program contains harmful instructions does not exist. If such technology is to be developed, it will likely center around automatically proving certain formal properties of programs. Functional languages are particularly suitable for writing programs which can be proven to have certain formal properties. Compilers for functional languages have long been performing many static checks on programs; these checks could be extended to include security checks, and the results of checking could be incorporated into an executable program and automatically verified when a user runs the program [3]. In this way, abstraction barriers could vary according to this security information: in the context of the above example, the user of an e-mail client could choose to automatically run only those executable attachments which are accompanied by verifiable security information.

The Power of Functional Programming

An example of a problem which incorporates both formal methods and reducing the modularity/efficiency tradeoff is my current senior thesis research.[2] The type inference approach to deforestation is unique among deforestation methods, as it relies on deriving properties of an arbitrary program rather than assuming that the program already has certain properties [1]. Developing a full implementation of type-inference based deforestation, as I am working on doing, will demonstrate that both type inference and deforestation are more useful than was previously imagined. I intend to continue doing work that uses formal methods to make functional programming more efficient and powerful, as this idea has implications that go far beyond my work this year.

References

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

[2] Hughes, John, "Why Functional Programming Matters", *Computer Journal*, v. 32, no. 2, pp. 98-107, 1989.

[3] Necula, George, "Proof-Carrying Code", <http://www.cs.berkeley.edu/~necula/pcc.html>, 1999

[2] described in my Previous Research Experience statement