Declarative Infrastructure for Automated Scientific Computing

Introduction

I apply techniques from the programming languages community within the scientific computing ecosystem, particularly in statistical and numeric computing.

The goal of my research is to promote independent development of mathematical and algorithmic solutions in scientific computing. Problems in this domain require a depth of scientific, mathematical, and computational expertise which are hard to find in combination in an individual. My research endeavors to separate the scientific computing problem so that experts from many fields may contribute independently without close collaboration with the others. This is hard. Problems in scientific computing often have high performance standards requiring tight integration between mathematical and algorithmic code.

In this document I explain my present and future work with the following structure. In Section 1 I present a toy example. In Section 2 I describe my previous work and how it brought me to my future work. In Section 3 I present motivation and background for my future research, declarative infrastructure for computational mathematics.

The importance of expertise for performance

Consider the following three solutions to the factorial problem [1]

```c
// Naive
int fact(int n){
    if (n == 0) prod = n;
    return prod * fact(n-1);
}

// Programmer
int fact(int n){
    prod = n;
    return lround(exp(lgamma(n+1)));
}

// Mathematician
int fact(int n){
    prod *= n;
    return lround(exp(lgamma(n+1)));
}
```

The left is the standard solution, written by a standard researcher. In the second an expert programmer replaces recursion with iteration to avoid a large call stack. In the third a mathematician uses the relation $n! = \Gamma(n+1)$ and the built in log-gamma routine.

The second and third solutions are substantially more efficient but require knowledge accessible to only a few experts. Fortunately, expert programmers have built compilers with optimizations that automatically transform the first solution into the second, increasing performance for many. Through compilers the programming languages community has expressed and automated their expertise so that it may benefit others.

My research leverages this same compiler technology to automate and disseminate mathematical and computational expertise. In the above toy example, mathematical knowledge transforms an $O(n)$ operation into an $O(1)$ operation. This benefit is common; proper understanding of mathematics often confers substantial speedups in scientific problems. Automation in these domains result in more efficient solutions, code reuse, and a lower barrier to entry. We make some observations:

- Few researchers know both forms of expertise. It is rare to find someone who knows both tail-call optimization and the gamma function.
• Complex problems often require several domains of expertise.

• Automation comes more naturally to programmers than to people of other domains. I.e. it is unsurprising that the second solution has been automated but not the third.

It is difficult to find mathematical experts who are able to build systems to automatically manipulate and generate high performance code. Programming language and scientific/mathematical theories are rarely taught to the same researchers. To this end I focus on declarative solutions that enable the separate expression of mathematical logic and programmatic control.

Previous and Current Work

I have applied my expertise to scientific computing problems at increasing levels of infrastructure. In analogy with the Fibonacci problem, there are three levels of infrastructure.

1. One can apply expertise to write down high quality solutions. (listing 1b, 1c)

2. One can programmatic ally express expertise and build a compiler so that expertise may be automatically applied (code transformations like tail call optimization in compilers like gcc)

3. One can build infrastructure to allow domain experts to express their expertise and automate the generation of compilers.

My masters work is about the first. My PhD work is about the second. My future work will be about the third. Past work is described below and future work in the next section.

Originally I applied expertise in physics and numerics to directly solve a single problem, in particular the effects of uncertainty in weather prediction on the power grid [2, 3, 4]. Subsequently I encoded my expertise into several micro-languages and compilers for stochastic modeling [5, 6], matrix expressions [7], and augmented an array computing framework [8] with concurrent GPU and distributed MPI primitives. Each project expresses the expertise of a single field in isolation (statistics, linear algebra, parallel computing.) My thesis work connects several such orthogonal projects to solve multidisciplinary problems, primarily in distributed numerical linear algebra. My future work is about how the process of mathematical compiler-building may be generalized.

I found that I was able to reuse much of my work between different micro-languages by using term rewrite systems. Rewrite rule systems enable the separate expression of mathematical transformations from programmatic control flow. For example the following relations in linear algebra

\[ X^T \rightarrow X \text{ if } X \text{ is symmetric} \]
\[ X^{-1} \rightarrow X^T \text{ if } X \text{ is orthogonal} \]

can be expressed declaratively by a mathematician with the following Python code.

```python
rl1 = rewriterule(X.T, X, wilds=[X], condition=Q.symmetric(X))
rl2 = rewriterule(X.I, X.T, wilds=[X], condition=Q.orthogonal(X))
```
An algorithmic programmer can then use control strategies to specify that these rules should be applied
together (multiplex), throughout an entire expression (even within subexpressions) (top_down) and should
applied exhaustively until there is no change (exhaust)

```plaintext
fn = exhaust(top_down(multiplex(rl1, rl2)))
```

In this way the mathematical code and the graph traversal code are written separately by a domain scientist
and computer scientist respectively without collaboration.

**Rewrite Rule Infrastructure for Domain Specific Compilers**

Mathematical experts are rarely trained to create compilers. The understanding of control flow and program-
matic graph transformations is unfamiliar. Declarative programming techniques like rewrite rule systems
separate the expression of mathematical logic from the specification of control flow. Mathematicians and
algorithmic programmers can develop each half separately without close collaboration. Simultaneous expertise
is not required.

Rewrite rule systems are used successfully in popular computational projects like Spiral [9] within the
computational community and have mature theoretical backing from implementations like Stratego/XT [10]
within the compilers community. However no broad rewrite rule framework has taken off in the open source
computational science community.

My current work successfully uses this technology in the Python language, a popular glue language in scientific
communities. It is embedded into the widely distributed SymPy project[11], where it has received considerable
attention and continued support. This work can be extended by developing new domain optimizations, new
control functionality, and better internal infrastructure. Much of the existing work should be separated off
to a separate project for general logic programming [12]. A set of strategies were implemented to mirror
those in the Stratego project; these should be developed to create complex control patterns (e.g. dynamic
programming) that can be used to more effectively search the space of potential algorithms.
References


