Metaprogramming for the Creation of Scientific Software

Gal Berkooz, Paul Chew, Jim Cremer, Rick Palmer and Richard Zippel
Cornell University
Ithaca, NY 14853

Abstract

A major cost in scientific computing is the creation of software that performs the numerical computations. This paper presents preliminary results on research to build a framework for automating the construction of numerical solvers for differential equations. Within this framework, the scientific computing problem is described using a very high level programming language that captures the original differential equations in a natural fashion. A sequence of code "transformers" are used to gradually refine the high level description of the problem into a concrete, executable form. Numerical techniques like the finite element method, the spectral method and the Crank-Nicolson discretization scheme are encoded in these transformers and once so encoded can be applied to a wide variety of different problems.

Introduction

One of the most difficult aspects of large scale scientific computations is generating the programs that perform the numerical computations. These programs often involve extensive, intricate mathematical computations whose coding can be quite error prone. In certain domains, such as turbulent fluid flow, it can take years to write efficient scientific software. These problems are exacerbated when parallelism is involved. This problem has been recognized by a number of authors [Abelson and Sussman, 1989, Abelson et al., 1989, Cook, Jr., 1990, Hilfinger and Colella, 1989, Kant, 1991, Kant et al., 1991, Steinberg and Roache, 1985, Wang, 1988, Wang et al., 1984].

We are developing a framework to automate the generation of this type of numerical software. Our approach translates the problem of code generation into one of program transformations. This enables us to use existing programming language, program transformation and compiler technology. The intermediate structures produced are combinations of mathematical statements and programming constructs. These novel structures allow us to incorporate deductive reasoning and type inferring technology to ensure the resulting numerical programs are correct. As with many of the previous approaches we also use symbolic computing, although in a slightly different fashion.

Burgers’ Equation

To illustrate our approach, we consider the following very simple example; a driven version of Burgers’ equation:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2} + f(x, t),$$  \hspace{1cm} (1)

where $f(x, t)$ is a driving force (provided numerically) and where $u$ satisfies the periodic boundary condition $u(t, x) = u(t, x + L)$ and the initial condition $u(0, x) = u_0(x)$.

We want to translate this mathematical problem into executable code. Several important issues are stated only implicitly in the mathematical formulation of the problem. To create a computer program for the problem these issues must be made explicit. The most important of the unstated issues is the “type” of $u$: $u$ is a function from time and space into $\mathbb{R}$. To capture the periodic boundary condition $u(t, x) = u(t, x + L)$ we make the spatial domain be the periodic interval $[0, D)$ which we denote by $D = \text{Per}(0, L)$. Thus $u$ is an element of a Hilbert space:

$$u \in \text{Hilb}([0, T] \times D \rightarrow \mathbb{R}).$$  \hspace{1cm} (2)

The top half of Figure 1 illustrates these domains graphically. We find that this graphical representation gives the clearest image of the progress of the code transformations. Most likely, this graphical representation of $u$’s Hilbert space will be central to the user interface of the code generation system we are building.

The original differential equation problem (1) is given in the bottom half of Figure 1 in a readable version of the data structures that our system manipulates. To distinguish the programming fragments (which are executable) and mathematical expressions (which need further conversion), the typewriter font is used for all programming expressions.
The declare statement indicates the mathematical type of \( u \), i.e., the Hilbert space mentioned earlier. In this statement we have used a dot (\( \cdot \)) to indicate a variable that we do not want to name, but which conventional mathematical notation requires. In this case it would be more natural to use the syntax of (2), but then an additional statement would be needed to indicate where the arguments to \( u \) are placed. Notice that this declaration provides information about the analytic character of \( u \).

The “for all” (\( \forall \)) statements are interpreted mathematical expressions that represent sets. Thus, \( \forall (t \in [0, T]) \land (x \in D). \{ \frac{\partial u(t, x)}{\partial t} + u(t, x) \frac{\partial u(t, x)}{\partial x} = \frac{\partial^2 u(t, x)}{\partial x^2} + f(x, t) \} \), is interpreted as a set of “incremental constraints” on \( u \) that hold at each point in \([0, T] \times D\). Notice that the mathematical types of \( t \) and \( x \) may be determined from their domains \([0, T]\) and \( D \) respectively. The occurrence of \( t \) and \( x \) in the denominator and derivatives is not a reference to points in \([0, T]\) and \( D \), but rather to directions. This is an issue that conventional mathematical notation obscures.

The first argument to the “solve” is a continuously infinite set of constraints on \( u \). The second argument indicates that \( u \) is to be determined over its entire domain of definition subject to the constraint (1). The first solve expression in Burgers1 can be easily converted to an assignment statement later.

The “program” given in Figure 1 contains a complete description of the desired computation. It accurately and succinctly conveys the desires of the mathematician. What it does not contain is a specification of how to accomplish the computation. Our goal is to convert this mathematically intuitive “program” into an executable code. In spirit, our approach is similar to that of a compiler, however there are differences that will become apparent as we proceed.

The most important difference is that unlike a compiler, we expect the programmer to specify the techniques to be used to convert the mathematical program into a more concrete, executable one. The techniques we are currently considering can be viewed as successively discretizing the Hilbert space of \( u \). Initially, the Hilbert space is consists of functions from a continuous domain to \( \mathbb{R} \) and as each dimension is discretized, the domain of the the function becomes continuous in fewer dimensions until ultimately, the domain of \( u \) is a discrete set of points. At this point, \( u \) can be viewed as a array of values (one for each point in the domain of \( u \)) and the constraints are just numerical equations that are to be solved.

The discretization process is specified by the user who indicates which numerical method is to be used in each dimensions. The system then applies the numerical method to the “program” to produce a new program with fewer continuous variables. As each discretization method is applied the program becomes less mathematical and more concrete, until finally we are left with a conventional numerical program.

For this simple one dimension problem, four steps are taken to generate the concrete program. First, time is discretized into equal sized steps. Second, a dual domain to the spatial domain is introduced, the frequency domain which is related to the spatial domain via the Fourier transform. The program is optimized so that some computation is done in the frequency domain and some in the spatial domain. Third, the spatial domain (and its dual frequency domain) are discretized. At this point, the constraints are discrete, but are still expressed as equality constraints and not assignment statements. Finally, the concrete code is generated.

In this example, some code motion and and optimization is performed between the second and third steps. In general, optimizations, parallelizations and more sophisticated mathematical transformations could be performed throughout the process. This should be contrasted with the standard approach in compilers, which only have access to the final code.

The “domain” of which \( u \) is an element plays a central role in our organization. The programmer’s intent is conveyed by indicating how the domain should be changed. Thus, the first step in the code generation process, discretization of time, is described by indicating that \( u \)’s domain should be changed.
For this example, we divide the time interval into equal steps of length $\Delta t$. This converts the continuous interval $[0, T]$ into a set of the discrete time values $\{0, \ldots, T/\Delta t\}$. This effect is graphically represented in the passage from Figure 1 to Figure 2. Denoting the time discretized version of $u$ by $u_{\text{time}}$, we have $u_{\text{time}} \in \text{Hilb}\{\{0, \ldots, T/\Delta t\} \times D \rightarrow \mathbb{R}\}$. It is conventional to write $u_{\text{time}}(n, x)$ as $u^{(n)}(x)$.

The effect of this transformation on the code is relatively straightforward. Instead of having a continuous set of constraints on $u(t, x)$ for all $t \in [0, T]$, we have a discrete set of constraints on $u^{(n)}(x)$.

The differential equation itself is transformed by a combination of two different techniques. First, the $u_t$ term is discretized as $\frac{u^{(n+1)} - u^{(n)}}{\Delta t}$, then the linear terms are transformed by the Crank-Nicolson approach: $g(u(t, x)) \rightarrow \frac{g(u^{(n+1)}(x)) + g(u^{(n)}(x))}{2}$, while a second order Adams-Bashforth technique is used for the non-linear terms: $h(u(t, x)) \rightarrow \frac{1}{2} \left[ 3h(u^{(n)}(x)) - h(u^{(n-1)}(x)) \right]$. Applying these transformations to the previous code module gives the code in the bottom half of Figure 2. Notice that the `solve` statement has been moved inside the loop over the discrete time values. That is, we have used a serializing transformation that converts $\text{solve}(\forall n \in \{0, \ldots, T/\Delta t\} \land (x \in D). \{\ldots\})$ to $\text{solve}(\forall n \in \{0, \ldots, T/\Delta t\} \{\text{solve}(\forall x \in D. \{\ldots\})\})$. This transformation indicates that a parallel system of equations can be solved one at a time. Needless to say, this transformation is not always valid, though when it is valid can be determined automatically. In this particular case, the serialization transformation is a component of the time discretization method.

The next step is to take the Fourier transform of the program. The effect on the domains is a bit more complex: a frequency domain, which we denote by $\tilde{D}$, dual to the spatial domain, must be created (see Figure 3). The dual of $u^{(n)}$ is denoted by $\tilde{u}^{(n)}$. It is an element of the Hilbert space $\tilde{u}^{(n)} \in \text{Hilb}\{\{0, \ldots, T/\Delta t\} \times \tilde{D} \rightarrow \mathbb{R}\}$.

At this point it is also appropriate to isolate $\tilde{u}^{(n+1)}$ on the left hand side of the equation. This illustrates the dependence of our techniques on symbolic computation. Notice that an inverse Fourier transform is needed at the end of Burgers3 to ensure that the behavior of the program was unchanged from Burgers2.

It is worth noticing that a certain amount optimization of common subexpression elimination and strength reduction can be accomplished at this stage. Because of the compactness of the form of the code in Burgers3, it is significantly easier to do perform these optimizations now than waiting until executable code is available as is done in compilers.

At this point it is easy to recognize that the terms $\mathcal{F}\{u^{(n)}u^{(n)}\}$ and $\mathcal{F}\{u^{(n-1)}u^{(n-1)}\}$ are closely related and only one needs to be computed, the other having already been calculated during an earlier pass through the loop. Our techniques allow such optimizations to be efficiently detected. These types of optimizations...
Burgers3 :=
declare \( u^{(1)}(\cdot) \in \text{Hilb}(\{0, \ldots, T/\Delta t\} \times D \rightarrow \mathbb{R}) \);
declare \( \hat{u}^{(1)}(\cdot) \in \text{Hilb}(\{0, \ldots, T/\Delta t\} \times \hat{D} \rightarrow \mathbb{R}) \);
solve(\forall x \in D. \{ u^{(0)}(x) = u_0(x) \}, \forall x \in D. \{ \hat{u}^{(0)}(x) \});
solve(\forall \omega \in \hat{D}. \{ \hat{u}^{(0)}(\omega) = F\{u_0(x)\}, \forall \omega \in \hat{D}. \{ \hat{u}^{(0)}(\omega) \});
for \( n \) in \{0, \ldots, T/\Delta t\} {
solve(\forall \omega \in \hat{D}. \{ \}
\left( \frac{1}{\Delta t} - \frac{\omega^2}{2} \right) \hat{u}^{(n+1)}(\omega) \right)
\left( \frac{1}{\Delta t} - \frac{\omega^2}{2} \right) \hat{u}^{(n+1)}(\omega)
\frac{1}{2} \left[ \frac{2}{\Delta t} \hat{u}^{(n)}(\omega) - \omega^2 \hat{u}^{(n)}(\omega) + F\{f(x, (n+1)\Delta t) + f(x, n\Delta t)\}
\right]
- \frac{3}{2} F \left\{ \hat{u}^{(n)}(x) \frac{\partial \hat{u}^{(n)}(x)}{\partial x} \right\} + \frac{1}{2} F \left\{ \hat{u}^{(n-1)}(x) \frac{\partial \hat{u}^{(n-1)}(x)}{\partial x} \right\}
\}
\forall \omega \in \hat{D}. \{ \hat{u}^{(n+1)}(\omega) \};
\hat{u}^{(n+1)}(x) \leftarrow F^{-1}\{\hat{u}^{(n+1)}(\omega)\};
}

Figure 3: Fourier transformed program

can be particularly valuable for problems involving more complicated PDE's and for producing code to be run on a parallel machine.

The particular optimization that needs to be applied depends upon recognizing structures like

\begin{verbatim}
for \( n \) in ... {
... \( f(n) \) ... \( f(n+1) \)
}
\end{verbatim}

where \( f \) is a computationally expensive expression. When constructs like this are encountered, the repeated computations are cached in other variables as follows

\begin{verbatim}
for \( n \) in ... {
\( f_n \leftarrow f(n) \);
... \( f_n \) ... \( f_{n+1} \)
}
\end{verbatim}

where the left arrow indicates assignment. When applied to Burgers3, this technique yields the code in Figure 4.

Similar types of transformations can be applied to Burgers4 to discretize the spatial/frequency domains, and convert the remaining solve into an assignment. For brevity these details are omitted.

Implementational Technique

The set of transformations described here require a number of different facilities: code transformations, symbolic computing and reasoning about domains. These disparate needs were met by implementing the transformations in Common Lisp and using the symbolic computing substrate Weyl [Zippel, 1990].

The transformations themselves were implemented in a rather ad hoc fashion. Representations were developed for the code modules, including the domains and any parameters that arise. As we gain more experience with the transformation structure we expect that certain idioms and patterns of use will arise that can be captured. However, the complexity of some of the transformations (e.g., the Crank-Nicolson/Adams-Bashforth transformation used for time) encouraged us to make a complete programming language available to the transform writer.

Unlike most computer algebra systems, one does not use Weyl via special language. Instead, Weyl extends a high level programming language, in this case Common Lisp, to have symbolic computing facilities. Thus we did not have to sacrifice the advantages of Common Lisp for the computer algebra facilities. In addition, Weyl's domain theoretic organization pro-
Figure 4: Optimized, Fourier transformed program

Provided the basis for the construction and manipulation of Hilbert spaces. In the future, we expect domains to play a role in determining which discretization methods are appropriate and gluing together solutions from different differential equations. Weyl's concept of domains is a descendent of the ideas in Axiom [Jenks and Sutor, 1992].

It should be noted that the transformations themselves are "equivalence" transformations on the program. If the original "program," which is provided by the physicist, is correct then final program should also be correct. We found that the transform philosophy made it much easier to understand what the structure and role of the transformation modules should be. It also clarified the modularity of the transformations and indicated how the transformation modules could be reused and recombined for different problems.

In the previous section two different transformers were discussed: (1) a uniform step size, Crank-Nicolson/Adams-Bashforth discretization method and (2) a Fourier transform method. Neither of these methods was designed especially for Burgers' equation. They could be combined with other transformation methods to produce more sophisticated solvers. For instance, the Crank-Nicolson/Adams-Bashforth method could be combined with a wavelet based spatial discretization method, or the Fourier decomposition transformations could be combined with a Runge-Kutta time advancement strategy. Further any of these combinations could be easily applied to other differential equations to quickly produce runnable code. We feel that the ability to reuse different transformation modules is significant.

The language of Burgers1 should be viewed as a very high level scientific programming language. Thus far we have only used it to state problems of solving differential equations. It will be interesting to see how it evolves as it is used for problems whose final result is more complex—results that involve statistical averages or convolutions of numerical solutions, for instance.

Conclusions

We feel this research combines several key ideas, which are listed below.

- High level programming language that mixes mathematical and programming constructs.
- Use of domains as an organizing principle.
- Interpretation of PDE's as the more computational concept of "continuous" constraints.
- Encapsulation numerical techniques as code transformers.
- Use of compiler optimization techniques on high level programming structures.

We feel the most important idea is the mixed mathematical/computational programming language that is used is the most important idea. Its use of mathematical domains (Hilbert spaces) as a "type system" provides the a powerful framework describing scientific computations and their transformation to effective programs.

There are several major areas where future research needs to be pursued. First, the mixed mathematical/computational programming language used throughout the code transformations needs to be formalized. The semantics and types of variables must be understood more thoroughly. In most earlier work, semanticists were free to design the programming language or correct language details that were at variance with their logical formalisms. The language we are using makes use of a fair amount of pre-existing mathematics, which does not necessarily match the current logical formalisms. Reconciling these two perspectives, the logical formalism of programming languages and the mathematical formalism of partial differential equations, appears to be challenging for both sides.

Second, we expect a large number of transformation methods to be created that discretize continuous mathematical programs. Each of these transformation methods should be able to decide whether or not they are applicable to a particular code module. This would allow the user to somewhat imprecisely specify how the code mode should be implemented, and the system would suggest only those transformation modules that are applicable. To support this process some deductive reasoning mechanism like Nuprl [Constable et al., 1986] or Ontic [McAllester, 1987] is needed and is currently being explored.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014-88-K-0591, by ONR Grant N00014-89-J-1946, NSF Grant 89006137 and AFOSR grant AFOSR-91-0328 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

References


