From equations to code: Automated Scientific Computing

Andy R. Terrel
Texas Advanced Computing Center, University of Texas at Austin
aterrel@tacc.utexas.edu

Traditionally, scientific programming progresses from equations to algorithms and finally to code given to the compiler. This strategy discards both the expressive language scientists use and the details required to optimize the code. Today, however, many projects are using domain specific languages (DSL) to give their users the ability to work directly with the equations. This process also allows domain specific optimizations which often speed up codes by many orders of magnitude. In this article, we give a small demonstration of this model of programming using a simple numerical integration example.

1 A simple example

Let us examine numerical integration of symbolic and discrete functions in one dimension as a simple example. Consider the following integral:

$$\int_0^1 (cos(x) * u(x) + sin(x)) * dx$$

where $u(x)$ is a function defined in your code. In the traditional style of coding, this simple equation would be a loop evaluating the equation at quadrature points. A programmer may come along and try to optimize the code a bit. A quick optimization would be to evaluate the integral of $sin(x)$, making the equation:

$$0.45969769 + \int_0^1 cos(x) * u(x) * dx$$

The code is still slow – after all $cos(x)$ takes many cycles to evaluate. The programmer might note that the procedure only evaluates the $cos(x)$ at the quadrature points and decide to substitute the values, resulting in something like:

$$0.45969769 + 0.48887694 * u[0] + 0.35239291 * u[1]$$

supposing that only two quadrature points are used. The programmer might be quite proud of the work; these small optimizations speed up this portion of code by a factor of 100 in the C programming language.

It is not hard to imagine that at some point, the project will need a more accurate solution. Unfortunately, the original equation has been lost to a sum of numbers and array indexing. An automated solution allows the user to return to the first equation and generates the optimized code. For our example, the code would be as follows:

```
u = DiscFunc("u")
```

```
x = Symbol("x")
dx = Dom(x, 0, 1)

integral = (Func(cos(x), x)*u + Func(sin(x), x))*dx
select_quad_rule(num_pts=2, name="Gauss")
gen_file("ex1", [integral],  ["eval_gen"], ['u'])
```

The generated file would implement the integral, making use of the above optimizations. For our example, the generated code is as follows:

```
const unsigned int NUM_QUAD_PTS = 2;
const double QUAD_PTS[2] = {-0.57735026919, 0.57735026919};
const double QUAD_WTS[2] = {1.0, 1.0};

inline double eval_gen(double* u)
{
  double ret_val = 0.0;
  ret_val += 0.459697694131860;

  ret_val += 0.488876937571022*(u[0])
          + 0.352392910067197*(u[1]);
  return ret_val;
}
```

When the user returns later, they are able to manipulate the equations in a much more natural way, while still having a fast implementation. In the following, I will explain how to embed a DSL, a technique that is used in various advanced packages (see section 4). Our implementation is in Python but many other high level languages have been used for the same purpose.

2 IntGen: A simple language for integral generation

To generate the above example, we are going to develop a small language embedded in Python and then generate C code. By embedding the language in Python, we avoid writing our own language tools, e.g., a parser, and are able to take advantage of the numerous Python libraries. In fact, our example will utilize the symbolic algebra system SymPy [9] to handle our symbolic functions. The BNF grammar of this simple language is as follows:

$$
\begin{aligned}
\text{integral} &\rightarrow \text{function} * \text{domain} \\
\text{function} &\rightarrow \text{discrete\_function} \\
&\mid \text{symoblic\_function} \\
&\mid \text{function} * \text{function} \\
&\mid \text{function} + \text{function}
\end{aligned}
$$

We will also implement a small code generator for integral expressions defined in this simple language. The code generator will choose the algorithm for each specific expression and thus apply the motivating optimizations to all inputs.

This small example will make use of some advanced features within the Python language. These features, such as the default factory method for classes `__new__`, allow us to concisely and efficiently create immutable expression trees. The formal explanation of these features is beyond the scope of this article, but hopefully their use will be apparent in our simple example.

2.1 Defining the language

The first task for our language is to represent expressions. We do this with the base class `IntGenExpr` below.

```
class IntGenExpr (object):
    """Base Expression Object"""
    def __new__ (cls, *args):
        obj = object.__new__(cls)
        obj._args = args
        return obj
    @property
    def args (self):
        return self._args
```

Each expression will be a tree, similar to output from a parser, with the operations as the node and the argument tuple as the leaves. In order to keep the argument tuple immutable, we use the `property` decorator to implement a getter function and indicate that stored arguments are private with a leading underscore.

A keen eye may have noticed that we are using the `__new__` method for the constructor. This method is similar to the standard `__init__` method, but rather than returning itself as the instance of the object, the `__new__` method returns an object. Python uses this method to realize the factory pattern for all of its objects. We use it in our language to efficiently combine objects if possible. For example, if two symbolic functions are added, we should return one symbolic function rather than an expression tree of an `Add` with two symbolic functions as arguments.

Our language uses three types of functions: symbolic functions, discrete functions, and function expressions. We will use operator overloading to implement the grammar rules that would normally be extracted by a parser. Python overloads multiplication and addition with the methods `__mul__` and `__add__`, respectively. We pass the algebraic operators of the functions to their respective implementations and also return an integral object if multiplied by a domain.

```
class Func (IntGenExpr):
    """Base Function object, also the symbolic function wrapper"""
    def __new__ (cls, sym_arg, var):
        return IntGenExpr.__new__(cls, sym_arg, var)
    def __mul__ (self, other):
        if isinstance(other, Func):
            return Mul(self, other)
        if isinstance(other, Dom):
```

```
            return Integral(self, other)
    def __add__ (self, other):
        if isinstance(other, Func):
            return Add(self, other)
```

To simplify the code, we use instances of the base function objects as the symbolic function. The symbolic functions have two arguments: a SymPy expression and the variable of integration . The only argument for a discrete function is the name of the array; the algebraic operators are inherited from the above `Func` class.

To implement the additions and multiplications, we inherit from the `Func` class and accept two or more arguments.

```
class Add (Func):
    """Function addition"""
    def __new__ (cls, *args):
        if len(args) == 1: return args[0]
        elif len(args) < 1: return None
        if all(map(lambda a: type(a) == Func, args)):
            return Func(sum([a.args[0] for a in args]), args[0].args[1])
        return IntGenExpr.__new__(cls, *args)
```

As mentioned earlier, our factory method `__new__` will combine the terms if they are all symbolic instead of return the `Add` object. Rearranging the object using this factory pattern can be quite powerful for manipulating the user input into canonical forms or simplified code. This will reduce the complexity of the code generator. The `Mul` class has a similar implementation to `Add`.

Integrals and domains are implemented in the same fashion as functions. An integral has two arguments: the integrand, which can be any function expression, and a domain. A domain takes a variable of integration and the two end points.

2.2 Implementing the code generator

Now that we have a language embedded in Python, we need a code generator. Below, our implementation is outlined with code to generate the C syntax omitted.

```
def int_gen(integral, func_name, input_vars):
    sym, disc = split_func(integral.args[0])
    code = gen_fdec(func_name, input_vars)
    code += gen_symbolic(sym, integral.args[1])
    code += gen_disc(disc, integral.args[1])
    code += gen_fclose()
    return code
```

The first optimization we make is to split the integrand into purely symbolic terms and terms with discrete inputs that must be evaluated at the quadrature points. We then generate the two parts separately.

The generation of the symbolic terms can call SymPy to get the exact integration,

```
def gen_symbolic(sympy_arg, dom, ret_str="ret_val"):
    intgrl = sympy_arg.integrate(dom.args)
    return "  ret_str += %f;\n" %  intgrl.evalf()
```

To generate the discrete functions, each expression containing discrete functions needs to produce its C evaluation string at the quadrature points. Since the language is already embedded in Python, we are able to give each object an `eval_pt` method which returns a tuple of a number and a string for indexing the discrete objects. This method is the analogue of writing a typical visitor function for the expression tree. To make the example self-contained, we implement a low-order Gaussian quadrature code with the weights and points stored in two global arrays, `QUAD_PTS` and `QUAD_WTS`.

```
def gen_disc(func, dom):
    disc_eval = ""
    jac = (dom.args[2] - dom.args[1]) / 2.0
    shift = (dom.args[2] + dom.args[1]) / 2.0
    for i in xrange(len(QUAD_PTS)):
        num, array = func.eval_pt(i, jac * QUAD_PTS[i] + shift)
        if i != 0:
            disc_eval += "\n           + "
        disc_eval += str(jac*num*QUAD_WTS[i]) + "*(" + array +")"
    return "  ret_str += %s;" % disc_eval
```

To use the generated library in its current form, we call the generated code with a C code. A user needs to make sure the C arrays conform to the evaluation of the quadrature points. For our simple case, we provided global arrays in the generated library for the discrete function to be tabulated.

```
#include "ex1.h"

int main()
{
  int i;
  double u[NUM_QUAD_PTS], val;
  // Fill u(x) = x shifting domain to [0, 1]
  for(i=0; i< NUM_QUAD_PTS; ++i)
    u[i] = 0.5*QUAD_PTS[i] + 0.5;
  val = eval_gen(u);
  printf("val = %f\n", val);
}
```

While this example may seem overly simple, it lays the basic structure for embedding a DSL and automatically choosing different algorithms to generate. Numerous further optimizations, such as unrolling the loops, lifting coefficient weights, and combining like integrals, could be made as the integrals become more complex. Using these techniques the FEniCS project has sped up the evaluation of its integrals by up to 1000X [7].

The full code for IntGen can be viewed at [10].

3 Try it out at home

Using a DSL embedded in a high level language and generating optimized low level code takes advantage of the expressiveness provided by the high level language and the performance characteristics of the low level code. Low level optimization, such as loop unrolling and blocking, have become standard practice for producing high performance code. Automation, however, opens the realm of possibilities to high level optimization, including selecting different algorithms based on architectural specifications.

The successful development of several automated systems has followed a general pattern: the definition of the problem domain, a well defined interface to divide the problem domain, and optimizations within each subdomain of the problem.

In our example, we were able to stick with simple integration routines and thus make a straight-forward language with a surprisingly small amount of code. Often formulating the problem correctly is the hardest part of the process, but can open the system up to many more optimization opportunities.

Our interface consisted of some function calls and global arrays. There are certainly more sophisticated systems available, but keeping it simple helps the user. Most automated scientific computing projects create several layers of access. This allows experts to link into the internals and customize the system to their desire, a useful feature for the users who have a legacy code they want maintained with code generation.

Optimization techniques drive the creation process for the automated system. The high level abstractions open up the system making a larger range of optimizations possible. At very least, low level optimizations should be reproducible and the original source readable. Because scientific computing is a unique discipline, it is not unusual to implement optimizations that are too specific to be in a general purpose compiler.

4 Examples in the Wild

The number of examples from the community are growing rapidly. Where standard compilers are failing to generate code that speeds up consistently on newer architectures, automated scientific computing is gaining speed by using the domain specific optimizations. Below, a few well known examples are outlined.

4.1 FEniCS

The FEniCS project [5] provides software for automated solutions of differential equations. The FEniCS project has been very successful at creating advanced user codes that are readable, use domain specific optimizations, and generate competitive low level code.

A user writes the equation using the Unified Form Language (UFL) which is embedded in Python. The expression tree is passed off to one of two compilers that will generate Unified Form-assembly Code (UFC) which is a low-level C++ interface. A user can then efficiently solve the equation using DOLFIN which is a C++ library that manages meshes, assembles

the matrix equations, and calls off-the-shelf linear solvers. DOLFIN also includes a Python interface allowing the user to keep the productivity and readiblity advantages of the high level language with the speed of the low level generated code.

FEniCS has solved some very challenging scientific problems and users have contributed several application codes. For example, Rheagen is a code for applying different rheological models that can be formulated as PDEs. By extending the UFL language to understand methods of splitting stresses and stabilizing transport, Rheagen makes implementing rheology models from the last half century an afternoon's work.

4.2 FFTW

FFTW [2] is well known for automatically tuning its FFT algorithm for the specific machine it runs on. The strategies for building these algorithms are built through an embedded language in OCaml that generates fast C code. Through their DSL, they are able to generate optimized codes for any input length and real or complex data.

4.3 Spiral

The Spiral project [8] represents discrete signal processing operations in its own algebraic operator language and generates libraries in several languages. The code also incorporates a large search of the optimal algorithm given the particular operation and hardware where the library is to run. While it provides an FFT algorithm that is competitive with FFTW, it is able to optimize many more operations.

4.4 Tensor Contraction Engine

The Tensor Contraction Engine [1] is a DSL that compiles tensor contractions used in compuational chemistry. The language was developed to manage the enormous expansion of the tensor contraction required to achieve high performance. It has been able to take development time of chemistry codes from months to hours.

4.5 CUDA

NVIDIA's CUDA [6] language uses a similar approach to embed a DSL for GPUs kernel functions inside of traditional C++ code. This language has dramatically expanded the use of general purpose GPU computing. Before CUDA codes once had to use shader languages that did not fit well with standard computing structures to unlock the power of the GPU.

4.6 Theano

The Theano project [4] uses an expression language for multi-dimensional arrays in Python and generates code to acceleration the evaluation of the operation. Similar to the given example it is able to build a C or CUDA version of the expression to realize gains unattainable

by working directly in Python. Theano uses just-in-time compilation of the generated code and allows the user to keep working in Python.

4.7 Sailfish

The Sailfish project [3] implements a Lattice-Boltzmann language and generates code to the CUDA language. Using the same wrapping technique as Theano, the user is able to work in Python for the post-processing and visualization of the simulations. The simple interface gives users a simple way to create 2D and 3D simulations with numerous different equations.

References

[1] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, and Others, *Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models*, Proceedings of the IEEE, 93 (2005), pp. 276–292. Available from: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1386652`.

[2] M. Frigo, *A fast Fourier transform compiler*, Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation - PLDI '99, (1999), pp. 169–180. Available from: `http://portal.acm.org/citation.cfm?doid=301618.301661`.

[3] M. Januszewski and M. Kostur, *The sailfish project*. Available from: `http://sailfish.us.edu.pl/`.

[4] LISA lab at University of Montreal, *Theano - cpu/gpu symbolic expression compiler in python*. Available from: `http://deeplearning.net/software/theano`.

[5] A. Logg, *Automating the Finite Element Method*, Archives of Computational Methods in Engineering, 14 (2007), pp. 93–138. Available from: `http://www.springerlink.com/index/10.1007/s11831-007-9003-9`.

[6] Nvidia, *NVIDIA CUDA Programming Guide*, 2007. Available from: `http://developer.download.nvidia.com/compute/cuda/1\_1/NVIDIA\_CUDA\_Programming\_Guide\_1.1.pdf`.

[7] K. B. Ølgaard and G. N. Wells, *Optimisations for quadrature representations of finite element tensors through automated code generation*, ACM Transactions on Mathematical Software, 37 (2010). Available from: `http://dx.doi.org/10.1145/1644001.1644009`.

[8] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, *SPIRAL: Code Generation for DSP Transforms*, Proceedings of the IEEE, 93

(2005), pp. 232–275. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1386651.

[9] Sympy Development Team, *Sympy, python symbolic mathematics library.* Available from: http://sympy.org.

[10] A. R. Terrel, *Ignition: a numerical code generation library.* Available from: https://github.com/aterrel/ignition/tree/master/ignition/int_gen/.