C++ Templates as Partial Evaluation

Todd L. Veldhuizen*

Abstract

This paper explores the relationship between C++ templates and partial evaluation. Templates were designed to support generic programming but unintentionally provided the ability to write code generators and perform static computations. These features are accidental, and as a result their syntax and semantics are awkward. Despite being unwieldy, these techniques have become somewhat popular because they partially solve an important problem in scientific computing– how to provide libraries of domain-specific abstractions without performance loss. It turns out that the C++ template mechanism is really partial evaluation in disguise: C++ may be regarded as a two-level language in which types are first-class values and template instantiation resembles offline partial evaluation. That C++ templates have proven so useful underscores the potential importance of partial evaluation as a language feature.

1 Introduction

1.1 Overview

C++ templates are of interest since they solve some important performance problems in designing scientific computing class libraries (Section 1.2). Templates were intended to support generic programming, but accidentally provided the ability to perform static computations and code generation (Section 2). It turns out that the C++ template mechanism is a form of partial evaluation (Section 3); the experience of library developers working with templates may offer some useful insights about partial evaluation as a language feature (Section 4).

1.2 Motivation

Scientific computing requires many abstractions. Every subdomain has its own requirements, such as interval arithmetic, tensors, polynomials, automatic differentiation, meshes, and so on. For economic reasons, languages can only provide the few concepts common to all, such as floating-point numbers and arrays. In the past, people requiring more than the limited abstractions provided by mainstream languages have developed domain-specific languages (DSLs) for sparse arrays, intervals, automatic differentiation, adaptive mesh refinement, and others. Such languages are not ideal solutions: they tend to have short life-spans due to limited support and portability, suffer from a lack of tools (particularly debuggers), and it is usually impossible to use two DSLs in the same source file.

With the advent of languages such as C++ and Fortran 90 which provided object-oriented features and operator overloading, it has become possible to create abstractions *using the language itself*, and have notations which resemble the mathematics being implemented. In recent years there has been a proliferation of libraries which provide abstractions previously implemented as DSLs: data-parallel arrays, sparse arrays, interval arithmetic, and automatic differentiation are the most prominent examples.

However, the code generated by such libraries tends to be naive. For example, array objects implemented using operator overloading in C++ were originally 3-20 times slower than the corresponding low-level implementation. This was not because of poor design on the part of library developers, but rather because the language forced a style of implementation which was grossly inefficient. These performance problems are commonly called the *abstraction penalty*; efforts to solve them have been many and ongoing.

One might think that a sufficiently smart optimizer would eliminate the abstraction penalty. However, compilers have difficulty because they lack semantic knowledge of the abstractions: they do not know that a given piece of code represents (for example) a sparse array operation; instead, they just see pointers and loops. Knowledge of the semantics is essential for doing appropriate optimizations. Efforts to describe the semantics of a class to compilers have been largely unsuccessful. Libraries tend to have layers of abstraction and side-effects which cause further difficulties for optimizers. Also, it is doubtful there is a general-purpose solution: every problem domain has its own optimization tricks and peculiarities.

A more promising approach is to construct libraries which both provide abstractions, and control how they are optimized. This concept has been called "active libraries" [5]. Such libraries handle high-level optimization themselves, leaving only low-level optimizations (register allocation, instruction scheduling, software pipelining) to the optimizer.

Meta-level processing systems such as Xroma [5], MPC++ [9], Open C++ [3], and Magik [6] provide one possible route to building active libraries. Such systems open up the compilation system and allow libraries to plug in their own translation modules. While these approaches are showing promise, a potential disadvantage is the complexity

*Extreme Computing Laboratory, Indiana University Computer Science Department, Bloomington Indiana 47405, USA. tveldhui@acm.org

of code which one must write: modern languages have complicated syntax trees, and so code which manipulates and generates these trees tends to be complex as well.

C++ templates may point the way to a more usable solution. Template techniques have been used to solve the performance problems of C++ for arrays and linear algebra, and several libraries based on these techniques are being distributed (e.g. POOMA [11], Blitz++ [20], and MTL [14]). The syntax used to implement these libraries is awkward. Partial evaluation offers hope for a cleaner syntax: there is a strong resemblance between templates and partial evaluation, so some mechanism based directly on partial evaluation might solve the abstraction penalty problem, and avoid the awkward syntax of template techniques. There is some precedent for this hope: at least one scientific computing project [15] reinvented the notion of two-level languages to provide a convenient notation for generating runtime library routines for parallelizing compilers.

So there is potential for fruitful collaboration: library developers need the technologies being developed by the partial evaluation community. Researchers in partial evaluation might benefit from the experience of library developers working with templates: there is a growing understanding of what features are useful for creating active libraries.

2 The capabilities templates provide

2.1 Generic programming

The original intent of templates was to support generic programming, which can be summarized as "reuse through parameterization". Generic functions and objects have parameters which customize their behavior.[1] These parameters must be known at compile time (i.e. must be statically bound). For example, a generic vector class can be declared as:

```
1   template<typename T, int N>
    class Vector {
        ...

5   private:
        T data[N];
    };

    // Example use of Vector
10  Vector<int,4> x;
```

The `Vector` class takes two template parameters (line 1): `T`, a *type parameter*, specifies the element type for the vector; `N`, a *nontype parameter*, is the length of the vector. To use the vector class, template arguments must be provided (line 10). This causes the template to be *instantiated*: an instance of the template is created by replacing all occurrences of `T` and `N` in the definition of `Vector` with `int` and `4`, respectively.

Functions may also be templates. Here is a function template which sums the elements of an array:

```
11  template<typename T>
    T sum(T* array, int numElements)
    {
        T result = 0;
15      for (int i=0; i < numElements; ++i)
            result += array[i];
        return result;
    }

20  // Example use
    double a[] = { 1, 2, 3, 4 };
    double a_sum = sum(a,4);
```

This function works for built-in types, such as `int` and `float`, and also for user-defined types provided they have appropriate operators (`=`, `+=`) defined. Note that in line 22, no template parameters are provided – the compiler infers the template parameter `T` from the type of `a`. Templates allow programmers to develop classes and functions which are general-purpose, yet retain the efficiency of statically configured code.

2.2 Compile-time computations

Templates can be exploited to perform computations at compile time. This was discovered by Erwin Unruh [17], who wrote a program which produced these compile errors:

```
23  unruh.cpp 10: Cannot convert 'enum' to 'D<2>'
    unruh.cpp 10: Cannot convert 'enum' to 'D<3>'
    unruh.cpp 10: Cannot convert 'enum' to 'D<5>'
    unruh.cpp 10: Cannot convert 'enum' to 'D<7>'
    unruh.cpp 10: Cannot convert 'enum' to 'D<11>'
    ...
```

The program tricked the compiler into calculating a list of prime numbers! This capability was quite accidental, but has turned out to be very useful. Here is a simpler example which calculates `pow(X,Y)` at compile time:

```
29  template<int X, int Y>
    struct ctime_pow {
      static const int result =
         X * ctime_pow<X,Y-1>::result;
    };

35  // Base case to terminate recursion
    template<int X>
    struct ctime_pow<X,0> {
      static const int result = 1;
    };
40
    // Example use:
    const int z = ctime_pow<5,3>::result;  // z = 125
```

In C++, `::` is a scope resolution operator: `A::B` means, "the symbol B in scope A." The first template defines a structure `ctime_pow` which has a single data member `result`. The `static const` qualifiers of `result` indicate that its value must be known at compile time. `ctime_pow<X,Y>` refers to `ctime_pow<X,Y-1>`, so the compiler must recursively instantiate the template for `Y`,`Y-1`, `Y-2`, ... until it hits the base case provided by the second template, which is called a *partial specialization*. C++ compilers include a pattern-matching system to select among templates; in general the most specialized template is selected. This pattern-matching aspect of templates results in a resemblance to logic-programming systems; the implementation of `ctime_pow` above resembles a logic-programming implementation of `pow`:

```
43  pow(X,Y) :- X * pow(X,Y-1).
    pow(X,0) :- 1.
```

[1] In generic programming, *generic function* means a parameterized function; this is a different meaning than in e.g. CLOS and Dylan.

Here is an array class which uses `ctime_pow` to calculate the number of array elements needed:

```
45      // Array which is the same length in
        // every dimension
        template<typename T, int Length,
           int Dims>
        class IsoDimArray {
50         // ...
           static const int numElements =
              ctime_pow<Length,Dims>::result;
           T data[numElements];
      }
55
        // A 3x3 array: will have 9 elements
        IsoDimArray<float,3,2> x;

        // A 3x3x3 array: will have 27 elements
60      IsoDimArray<float,3,3> x;
```

When the `IsoDimArray` template is instantiated, `ctime_pow` is used to calculate the array size required. This allows the array elements to be allocated on the stack, which is much faster than dynamic memory allocation. Similar template techniques can be used to find greatest common divisors, test for primality, and so on – all at compile time. As an extreme example, it is possible to implement a subset of Lisp (encoded in templates) which is "interpreted" at compile time [4].

2.3 Code generation

It turns out that control structures (loops, if/else, case switches) can be mimicked in templates. For example, the definition of `ctime_pow` (Section 2.2) emulates a `for` loop using recursion. These compile-time programs can perform code generation by selectively inlining code as they are "interpreted" by the compiler. This technique is called *template metaprogramming* [19]. Here is a template metaprogram which generates a specialized dot product algorithm:

```
61      template<int I>
        inline float meta_dot(float* a, float* b)
        {
            return meta_dot<I-1>(a,b) + a[I]*b[I];
65      }

        template<>
        inline float meta_dot<0>(float* a, float* b)
        {
70          return a[0]*b[0];
        }

        // Example use:
        float x[3], y[3];
75      float z = meta_dot<2>::f(x,y);
```

In the above example, the call to `meta_dot` in line 75 results in code equivalent to:

```
76      float z = x[0]*y[0] + x[1]*y[1] + x[2]*y[2];
```

Recursion is used to unroll the loop over the vector elements. The syntax for writing such code generators is clumsy. However, the technique has proven very useful in producing specialized algorithms for scientific computing. The MTL library [14] uses similar generators to construct fast, fixed-size kernels for use in linear algebra routines. By composing these kernels, MTL is able to provide linear algebra operations which are sometimes faster than the native libraries provided by hardware vendors. Similar generators are used by the Blitz++ library [21] to specialize algorithms for small, fixed-size vectors and matrices.

It is even possible to create and manipulate static data structures at compile time, by encoding them as templates. This is the basis of the *expression templates* technique [18], which creates parse trees of array expressions at compile time. These parse trees are used to generate efficient evaluation routines for array expressions. This technique is the backbone of several libraries for object-oriented numerics [11, 20].

2.4 Traits

The *traits* technique [12] allows programmers to define "functions" which operate on and return *types* rather than data. As a motivating example, consider a template function which calculates the average value of an array. What should its return type be? If the array contains integers, a floating-point result should be returned. But a floating-point return type will not suffice for all arrays (for example, complex-valued arrays).

The problem may be solved by defining a *traits class* which maps from the type of the array elements to a type suitable for containing their average. Here is a simple implementation:

```
77      // default behavior: T -> T
        template<typename T>
        struct average_traits {
80        typedef T T_average;
        };

        // specialization: int -> float
        template<>
85      struct average_traits<int> {
          typedef float T_average;
        };
```

An appropriate type for averaging an array of type `T` is given by `average_traits<T>::T_average`. This pair of templates encodes the behavior, "use the array element type for calculating averages, except use float for arrays of integers." Again, note the strong resemblance between this traits class and a corresponding logic-programming implementation:

```
88      average_type(T)   :- T.
        average_type(int) :- float.
```

Here is an implementation of `average`:

```
90      template<class T>
        typename average_traits<T>::T_average
        average(T* array, int N)
        {
            typename average_traits<T>::T_average
95             result = sum(array,N);
            return result / N;
        }
```

This version correctly handles arrays of integers, floating-point, and complex arrays.

Similar problems are constantly encountered in templated class libraries. Templates provide general-purpose rules for creating functions and classes; traits allow you to handle the many exceptions which arise.

3 Templates as partial evaluation

Partial evaluators [10] regard a program's computation as containing two subsets: static computations, which are performed at compile time, and dynamic computations performed at run time. A partial evaluator evaluates the static portion of the program and outputs a specialized *residual* program.

To determine which portions of a program may be evaluated, a partial evaluator may perform *binding time analysis* to label language constructs and data as static or dynamic. Such a labelled language is called a *two-level language*. For example, a binding-time analysis of some scientific computing code might produce this two-level code fragment:

```
float volumeOfCube(float length)
{
      return pow(length,3);
}

float pow(float x, int N)
{
      float y = 1;
      for (int i=0; i < N; ++i)
            y *= x;
      return y;
}
```

in which static constructs have been overlined. A partial evaluator such as CMix [1] would evaluate the static constructs to produce the residual code:

```
 float volumeOfCube(float length)
{
     return pow3(length);
}

 float pow3(float x)
{
     float y = 1;
     y *= x;
     y *= x;
     y *= x;
     return y;
}
```

Such specializations can result in substantial performance improvements for scientific code [2, 8].

3.1 C++ as a two-level language

C++ templates resemble a two-level language. Function templates take both template parameters (statically bound) and function arguments (dynamically bound). For example, the `pow` function of the previous example might be declared in C++ as:

```
template<int N>
float pow(float x);     // Calculate pow(x,N)
```

The static data (`N`) is a template parameter, and the dynamic data (`x`) is a function argument. To incorporate template type parameters into this viewpoint, we need to regard types as first-class values. For example, in a declaration such as

```
template<typename X, int Y>
void func(int i, int j);
```

we regard `X` as a *type variable*, as in ML. Since C++ is statically typed, type variables may only be statically bound. This point of view has a certain simplifying power: for example, one can view `typedef`s as declarations of type variables:

```
typedef float float_type;
```

can be regarded as equivalent to the (fictional syntax)

```
typename float_type = float;
```

3.2 Template instantiation as offline PE

Partial evaluation of programs which contain explicit binding-time information is called *offline* partial evaluation. Template instantiation resembles offline partial evaluation: the compiler takes template code (a two-level language) and evaluates those portions of the template which involve template parameters (statically bound values). For example, consider this template class:

```
template<int X>
struct ulam {
    static const int result =
      ulam<(X % 2 == 0) ? (X/2) : (3*X+1)>::result;
};

// Base case: X = 1
template<>
struct ulam<1> {
    static const int result = 0;
};
```

The syntax `A ? B : C` is C's equivalent to the functional *if A then B else C*. When `ulam<X>` is instantiated, the `const` qualifier on `result` requires the compiler to evaluate the right-hand side of the assignment at compile time. So it determines if `X % 2 == 0` (whether `X` is even). If true, it instantiates `ulam<X/2>`; otherwise `ulam<3*X+1>` is instantiated. In theory, this continues until the compiler hits the base case `X=1`. Whether this recursion terminates for all `X` is a well-known open problem. In C++, it is impossible to determine if a chain of template instantiations will ever terminate. For this reason, compilers place arbitrary limits on the depth of template instantiation chains.

In C++, the binding time of code is inferred from the binding time of data: if an expression is assigned to a statically-bound value, the expression must be statically bound. Templates in C++ only allow monovariant binding times; it is not possible to have data or code which is statically bound in one context, but dynamically bound in another. For example, standard library routines such as `pow` and `cos` cannot be used at compile-time and for practical applications this limitation is frustrating.

C++ does allow part-static, part-dynamic structures. For example, the class

```
class Example {
    static const int x = 5;
    int y;
};
```

contains both a statically bound member (`x`), and a dynamic member (`y`). (Note that the `static` keyword refers to `x` being shared among objects of type `Example`, and not to binding times). Mixed static-dynamic data structures have proven very useful; for example, they are the basis of the expression templates technique [18].

4 What can be learned from the C++ experience?

Asymmetry between the static and dynamic language is bad. In C++, the static and dynamic aspects of the language bear little resemblance to each other. The static version of the language is maddeningly limited: there are no floating-point numbers, no objects, and no side-effects. It might be desirable to have near-perfect symmetry between the static and dynamic languages, even to the extent of allowing side-effects at compile-time. For example, being able to do file I/O and issue console messages at compile-time would be very useful: a library could generate specialized code based on the contents of a data or configuration file, and issue compile-time errors and warning messages. Not being able to issue customized diagnostic messages in C++ has hurt the usability of template libraries.

Reflection and meta-objects would be useful. A common headache in using C++ templates is that there is no way to enforce constraints on template parameters. If users unwittingly violate constraints, the result might be a cryptic error message, or in the worst case, the program might crash mysteriously. A staged or multilevel language with some simple reflection capabilities could provide a straightforward way to enforce constraints on template parameters (an idea due to Vandevoorde [5]). For example, the `sum` function template of Section 2.1 assumes that the template type parameter `T` is some numeric type which may be initialized to `0` and has the operator `+=` defined. With reflection and staging, the `sum` function could examine the parameter `T` and issue a friendly diagnostic message if these requirements were not met.

There are open problems in reconciling multilevel capabilities with other language features. Templates interact with other language features in bizarre ways. Some might regard this as evidence of poor design, but some of these shortcomings point the way to interesting, possibly unsolved problems. How can static initialization be handled sensibly in a multilevel language, particularly one with relative binding times? Is it possible to provide a multilevel language which preserves separate compilation? (This has been an enormous headache for C++). How should object-oriented language features interact with multilevel language features?

First-class types are good. The ability to construct and manipulate types has proven extraordinarily useful in writing scientific C++ libraries. In particular, the traits technique– being able to write functions which operate on and return types– has proven very valuable.

Fixed rules for selecting among multiple templates are bad. In C++, the rules for matching templates are well-designed – they work 95% of the time. It is the other 5% which is annoying, since the rules are hard-coded into the language. One can sometimes trick the compiler into selecting particular templates, but sometimes not. In a multilevel language, it would be possible to implement these rules in the language itself while avoiding the overhead of systems such as CLOS which resolve multimethods at run-time. This would allow matching rules to be customized when necessary. Languages in the ML family, with their pattern matching syntax, might provide a natural mechanism for duplicating the template pattern matching features.

Explicit binding-time annotations are good. Good optimizers with inter-procedural analysis and procedure cloning are approaching the power of online partial evaluation. But undirected specialization is only marginally useful to library developers. Optimizing the trade-off between compile-time and run-time evaluation is tricky: some non-trivial scientific computing codes have no dynamic inputs. Most importantly, online partial evaluation implies an assumption that execution time is proportional to the *amount of computation*. In most scientific computing codes, the cost is in *data flow* between the caches and main memory. Online PE can generate residual code with "less computation," but so far not "smarter computation". To generate smarter code, one needs a predictive model of the hardware: its caches, pipelines, and so on. These are very difficult decisions to automate. For example, the Cray optimizer has roughly 10^8 possible settings of its optimization switches; finding the best settings for a given code requires tedious experimentation, even by experts. However, in the hands of a library developer who understands the hardware, explicit binding-time annotations can be a powerful performance tool, since compile-time computations can be used to rearrange the flow of the run-time computation for efficient cache use.

It is sufficient to label data with binding times. In C++ templates, binding-time annotations apply to data only – there is no way to label pieces of code as static or dynamic. The binding times of code constructs follows naturally from binding times of data. Although this may sound limited, in practice it has been sufficient to solve many important performance problems in C++.

5 Conclusions

C++ templates have acquired a reputation as being overly complex. In their defense, templates started as a simple mechanism, and developed gradually over a decade in response to experimentation and the needs of users. This incremental process contributed to their current state of disarray. However, this same process has resulted in a useful inventory of the capabilities which library developers require. Anyone developing similar mechanisms based on partial evaluation may benefit from examining this inventory.

C++ with templates may be regarded as a two-level language in which types are first-class, statically-bound values. Template instantiation bears a striking resemblance to offline partial evaluation. That templates have proven so useful in C++ is an encouragement for continued work on partial evaluation as a language feature. Languages incorporating partial evaluation may offer a way to provide generic programming, code generation, and compile-time computation via a single mechanism with simple syntax. In particular, research on explicit binding-time annotations, staging, and the relationship between partial evaluation and type systems could have many fruitful applications; developers of scientific computing libraries would benefit from language features like these.

6 Acknowledgments

This work was supported in part by NSF grants CDA-9601632 and CCR-9527130. I am grateful to Robert Glück for useful discussions about partial evaluation and templates, to Michael Ashley and Olivier Danvy for shepherding this paper, and to anonymous reviewers for many helpful suggestions.

References

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] Berlin, A., and Weise, D. Compiling scientific code using partial evaluation. *Computer 23*, 12 (Dec 1990), 25–37.

[3] Chiba, S. A Metaobject Protocol for C++. In *OOPSLA'95* (1995), pp. 285–299.

[4] Czarnecki, K., and Eisenecker, U. Meta-control structures for template metaprogramming. http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm.

[5] Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., and Veldhuizen, T. L. Generative Programming and Active Libraries. In *Proceedings of the 1998 Dagstuhl-Seminar on Generic Programming* (1998), vol. TBA of *Lecture Notes in Computer Science*. (in review).

[6] Engler, D. R. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages (DSL'97)* (October 15–17, 1997).

[7] Glück, R., and Jørgensen, J. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation 10*, 2 (1997), 113–158.

[8] Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.

[9] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., and Kubota, K. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Reflection'96* (1996).

[10] Jones, N. D. An introduction to partial evaluation. *ACM Computing Surveys 28*, 3 (Sept. 1996), 480–503.

[11] Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., and Williams, T. Array design and expression evaluation in POOMA II. In *ISCOPE'98* (1998), vol. 1505, Springer-Verlag. Lecture Notes in Computer Science.

[12] Myers, N. A new and useful template technique: "Traits". *C++ Report 7*, 5 (June 1995), 32–35.

[13] Nielson, F., and Neilson, H. R. *Two-Level Functional Languages*. Cambridge University Press, Cambridge, Mass., 1992.

[14] Siek, J. G., and Lumsdaine, A. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments* (1998).

[15] Stichnoth, J., and Gross, T. Code composition as an implementation language for compilers. In *USENIX Conference on Domain-Specific Languages* (1997).

[16] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices 32*, 12 (1997), 203–217.

[17] Unruh, E. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.

[18] Veldhuizen, T. L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[19] Veldhuizen, T. L. Using C++ template metaprograms. *C++ Report 7*, 4 (May 1995), 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

[20] Veldhuizen, T. L. Arrays in Blitz++. In *ISCOPE'98* (1998), vol. 1505 of *Lecture Notes in Computer Science*.

[21] Veldhuizen, T. L., and Ponnambalam, K. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools 21*, 8 (Aug. 1996), 38–44.