

Funk: a framework for functional style in C++

Thomas Hallock

September 24, 2002

Contents

| | | |
|----------|--|-----------|
| 1 | a Primer in Template Metaprogramming | 4 |
| 1.1 | templates | 5 |
| 1.1.1 | template specialization | 7 |
| 1.2 | typedefs | 9 |
| 1.3 | example | 9 |
| 1.4 | summary | 14 |
| 2 | normal C++ programming vs. template metaprogramming | 15 |
| 2.1 | data | 15 |
| 2.2 | execution | 17 |
| 2.3 | pattern matching | 17 |
| 2.4 | dominant paradigm | 18 |
| 3 | The Funk functional programming system | 18 |
| 3.1 | expression templates | 18 |
| 3.2 | lambda and apply type system | 20 |
| 3.3 | evaluating Funk expression templates | 22 |

| | | |
|-------|---|----|
| 3.3.1 | dealing with data | 22 |
| 3.3.2 | defining an expression template's <i>final type</i> | 24 |
| 3.3.3 | switching the type of fully applied lambda expressions | 25 |
| 3.3.4 | final types of expression templates | 25 |
| 3.3.5 | applying values to lambda expressions | 28 |
| 3.4 | an example Funk evaluation | 28 |
| 3.5 | building funk structures with function templates | 34 |
| 3.5.1 | using type resurrection and overloaded operators to to build expression templates | 35 |
| 3.6 | syntax | 37 |
| 3.6.1 | lambda expressions | 38 |
| 3.6.2 | partial application | 38 |
| 3.7 | shortcomings of Funk as a sublanguage of C++ | 38 |
| 3.7.1 | polymorphism | 38 |
| 3.7.2 | naming Funk objects | 38 |

Abstract

C++ can be thought of as a two-level language, one level deals with values and is evaluated at run-time, the other deals with types and is evaluated by the C++ compiler's template instantiation mechanism at compile-time. Both are turing complete, and both can be used to leverage the power of the other. This paper provides a primer on the techniques used to program for the type-level language. These techniques are collectively referred to as *template metaprogramming*, or *TMP*. TMP techniques are then compared to their data-level programming counterparts. TMP concepts are then used to explain the internals of a C++ functional programming framework, called *Funk*, that makes use of template metaprogramming.

1 a Primer in Template Metaprogramming

Template metaprogramming is a programming method that uses the type system of the C++ compiler to perform computations by processing template parameters.

The three main features of the C++ language used for TMP are:

- `template` definitions
- `typedefs`
- `structured data`

These three elements can be combined in some very interesting ways to exploit the type processing features of C++. The term “feature hijacking” is a good way of describing the modus operandi of TMP because it uses C++ features in ways unanticipated by their designers. Let us start with an overview of the features hijacked by TMP and the problems they were originally intended to solve.

1.1 templates

C++ templates were designed to provide a means to construct data structures and functions based on a form-like meta-definition. A template can take type names and compile-time constants (CTCs) as parameters. Compile time constants as template parameters are useful but can be avoided in a discussion of TMP basics.

It is important to understand that template definitions have no actual representation in the code produced by the C++ compiler. The actual structure or function that gets produced from a template is called an *instantiation*. An instantiation is handled like any other non-template function or structure during compilation and runtime.

To understand the need for templates, consider the following structure definition:

```
struct IntArray {  
    int x[128];  
    int index(int) // get the value of x at position i  
        {return *(x+i);};  
};
```

This statement defines a single type called ‘IntArray’ each of whose instances holds an array of integers. A template definition works differently. When the C++ compiler sees the text:

```
template<class T>  
struct Array {  
    T x[128];  
    T index(int i)  
        {return *(x + i);};  
};
```

it does not immediately define a structure type called `Array`. In fact, the C++ compiler does very little in response to this definition. All it needs to do is make a note of the template's name and parameter types. Instead of defining a structure, this template definition defines *how* to define a structure.

Template parameters are declared between the `'template<'` and `'>'` tokens. Every template parameter's name must be preceded with the keyword `class` or the keyword `typename`. For consistency's sake, we will use the keyword `class` for template parameter definitions. When all of a template's parameters are concrete typenames, the template is said to be *instantiated*.

The following types are instantiations of the above `Array<T>` created by providing a typename for `Array<T>`'s type parameter:

```
Array<int>
```

```
Array<char>
```

```
Array<bool>
```

and the more interesting recursive instantiation:

```
Array< Array<int> >
```

The compiler creates code for a template instantiation by copying the template definition and replacing all occurrences of the typename variable (which in the case of `Array` is only `T`) with the named type. The compiler then derives a unique name for this new type from the types of the template arguments. The naming scheme varies from compiler to compiler but usually consists of a concatenation of the template name and the names of the template arguments. For example, a compiler might name the instantiation `Array<int>` as `"Array0_int"`. This identifier is usually referenced only during the reporting of errors or during a debugging session.

1.1.1 template specialization

Template specialization is a recent feature of C++ that allows structure templates to be defined differently when instantiated with certain types or type patterns specified by the programmer. The syntax for a template specialization is very similar to a regular template definition, but the difference in meaning is very important. Suppose that we had a processor that had special optimizations for storing and referencing arrays of integers. To take advantage of this optimization, we could specialize our `Array<T>` for the case when it is instantiated with `int` with the following code:

```
template< >
struct Array < int> // specialization overrides the template
{
    // definition for Array<T> when T is int.
    int x[128];
    int index(int i) {fast_index(x,i);}
};
```

The `< int >` portion after the template's name specifies the types for which this specialization will be used. In addition to specializing for concrete type names, it is also possible to define a specialization that works with a class of templates instantiated with *any* type. In order to express classes of template instantiations it is necessary to use generic type variables that act as "wild cards" in a type pattern. These type variables are declared in the `template< ... >` syntax above the specialization's type name. This explains why the above specialization for `Array<int>` has the funny declaration-free `template< >` syntax ; it simply has no type variable declarations. If we wanted to have special routines that can work with matrices, we could make a specialization of `Vector<T>` that gets instantiated whenever its template parameter is another `Array<T>.`:

```
template<class T>
struct Array< Array<T> > {
    int index(int i) { return x[i]; }
```

```

int determinate () { ... }

std::pair<Array< Array<T> >,Array< Array<T> > > LU_decomposition() { ... }

Array< Array<T> > row_echelon_form() { ... }

Array<Array<int> > x
};

```

Similar functionality can be achieved through the use of inheritance, but this method avoids the use of virtual functions which tend to slow down run-time execution. Also the added functionality is obtained transparently with TMP, whereas when using inheritance, the added functionality would need to be requested explicitly (e.g. `new Matrix()` or `new MMXArray`.) OOP techniques can also be used to assist in TMP, this is explained in the discussion on type resurrection.

When no type variables are defined for a template specialization, it's called a *full* template specialization. When type variables are defined for use in the pattern specified after the specialization's name, it is called a *partial* template specialization.

Because it is possible to define multiple specializations for a template, it is also possible to define two specializations that could match the same instantiation, causing what might appear to be a specialization conflict. To resolve this issue, the standard for C++ states for the compiler to follow:

- Always check the specializations for a match before the base template is matched.
- Template specializations that are more specialized ¹ have a higher priority for matching than those that are less specialized.

¹In the C++ programming Language, Stroustrup[Str97] defines “more specialized” with the following: One specialization is *more specialized* than another if every argument list that matches its specialization also matches the other, but not vice versa.

A problem arises when two specializations are equally specialized. When this happens, the programmer cannot be certain which specialization will be instantiated. However, this situation does not arise often and can usually be avoided.

1.2 typedefs

`typedefs` are typically used to give simple names to more complex type names. They are the C++ type equivalent of an English synonym. For example, if I have a very long and complicated type name, an STL vector iterator, for example, I can give it a C++ type synonym using the `typedef` feature:

```
typedef std::vector<string>::iterator    itertype;
```

This creates a type synonym `'itertype'` for the Standard Template Library's `'vector<string>::iterator'`. Any mention of `'itertype'` is the same as mentioning `'std::vector<string> iterator'`.

In TMP, `typedefs` are used to define template metaprograms. Metaprograms are executed by referencing the `typedef`. By defining a `typedef` inside of a template specialization, `typedefs` can be defined recursively using type variables. This allows for the template metaprogramming equivalent of functions to be defined using `typedefs`. The TMP values that these TMP functions return are types.

1.3 example

As an exercise, let's look at how we can use the C++ compiler to compute Fibonacci numbers. For the compiler to be able to perform addition, we will need to build up the rules for addition in an axiomatic format. We begin by defining the set of natural numbers recursively, starting with a type representing the number 0:

```
struct ZERO { };
```

Next, we define a successor function:

```
template<class>
struct S { };
```

We can use `S<T>` and `ZERO` to express the entire set of natural numbers through recursive instantiations. While it is possible to instantiate `S<T>` with any type, we are going to instantiate `S<T>` only with other instantiations of `S<T>` or with the base type `ZERO`.

Now that we have a suitable definition of the natural numbers, we can define template metaprograms that can perform operations on them. All metaprograms must have an *activator type* that serves as an executor for the metaprogram. When this activator type is referenced, the compiler executes the metaprogram by instantiating the type that the activator type serves to define. C++'s awkward syntax for TMP definitions sometimes makes it difficult to determine their meaning. For clarification, consider the following generic definition of a TMP

```
template<class, ...>
struct name {
    typedef default type;
};
```

```
template<class A>
struct name<pattern<A> > {
    typedef result type;
};
```

Using the same identifier names in a pseudo language that is easier to understand, we might re-define the above TMP as:

```
name data = if (matches data (pattern A)) then result else default
```

All activator types in this paper are called `type` and are declared as typedefs inside of template specializations. Throughout compilation, type variables are immutable. Because of this, all looping in template metaprogramming must be performed through recursion. So, in order to implement basic arithmetic, we can use the definitions provided by the Peano axioms, which are recursive. We define `sum` like this:

```
template<class,class> // base template definition for later specializations.
struct SUM { };      // This non-specialized version should never get matched

template<class T>
struct SUM<T, ZERO > { // specialization for adding any value to zero.
    typedef T type;    // return the first template parameter for the identity.
};

template<class T0,class T1>
struct SUM<T0,S<T1> > { // specialization for adding any value to any non-zero value.
    typedef SUM<S<T0>,T1>::type type; // use the recursive definition of sum.
};
```

The above partial specialization is a template metaprogram version of the following Peano axiom:

$$\text{sum}(x,0) = x$$
$$\text{sum}(y,s(x)) = \text{sum}(s(y),x)$$

The first definition of the `SUM` type is required even though it is never used. This is because the C++ language requires all specializations to have a preceding regular, non-specialized definition.

We now have the facilities to define the Fibonacci TMP:

```

template<class>
struct FIB { }; // base template definition for later specializations.

template<class T>
struct FIB<S<S<T> > >{ // match any number two or greater

// use the recursive Fibonacci definition:
// fib(s(s(x))) = fib(x) + fib(s(x))
    typedef SUM< FIB<T>::type,
                FIB<S<T>
                >::type>::type type;
};

template<>
struct FIB<ZERO> { // catch zero
    typedef ZERO type; // fib(0) = 0
};

template<>
struct FIB<S<ZERO> > { // catch one
    typedef S<ZERO> type; // fib(1) = 1
};

```

Now that we have defined all the TMPs needed to generate Fibonacci numbers, we need a way to output the results. If we really wanted to prove that the numbers were computed during compile time, we could define ONE so that any instantiations would lead to compiler errors, and then determine the result by counting the compiler errors. However, a more user-friendly and error-free way would be to output the count of recursive

instantiations to standard output. The following function template definition counts the number of recursive instantiations of `S<T>` and returns the count:

```
int count (ZERO) { //catch if we are instantiated with ZERO
    return 0;      // return zero.
}

template<class T>
int count (S<T>) { // catch if we are instantiated with S<T> (any non-zero type value)
    return 1 + count (T()); // add one to the result and call with an instantiation of T.
}
```

in the latter definition of `count`, the expression `T()` creates a new temporary instance of type `T`. The only reason we need to do this is because the version of `count` that gets called depends on the type of its parameter. The simplest way to specify the type of `count`'s parameter is to create a new instance of `T` every time `count` is called.

The `main` function declares variables `f1` through `f6`, each having a type corresponding to its order in the Fibonacci sequence. It then shows the output of the `count` function when applied to each variable.

```
int main () {
    FIB<ZERO >::type f0;
    FIB<S<ZERO> >::type f1;
    FIB<S<S<ZERO> > >::type f2;
    FIB<S<S<S<ZERO> > > >::type f3;
    FIB<S<S<S<S<ZERO> > > > >::type f4;
    FIB<S<S<S<S<S<ZERO> > > > > >::type f5;

    cout << count (f0) << endl;
```

```
    cout << count (f1) << endl;

    cout << count (f2) << endl;

    cout << count (f3) << endl;

    cout << count (f4) << endl;

    cout << count (f5) << endl;

}
```

upon execution, the program will output:

```
0
1
1
2
3
5
```

1.4 summary

The C++ features described in this section can be summarized in what I call the template metaprogramming hijacking table. The table lists programming language features of C++ on the left and their alternate use as a programming language feature in template metaprogramming on the right.

C++ features hijacked by Template Metaprogramming:

| C++ feature | TMP use |
|------------------------------------|----------------------|
| <code>struct</code> | function declaration |
| <code>typedef</code> | function definition |
| <code>typedef</code> reference | function call |
| <code>template</code> parameters | data variables |
| <code>structured data</code> types | data |
| template specialization | pattern matching |

From the chart, it becomes evident that most techniques used in TMP have a C++ data-level counterpart. An interesting tangent re-write of the C++ compiler might try to unite the two levels of this language so that their syntaxes match more closely[Vel02]. This would provide for greater utilization of the C++ compiler as a code preprocessor and optimizer by allowing the programmer to specify in a familiar syntax the data that can be computed at compile-time instead of run-time.

2 normal C++ programming vs. template metaprogramming

2.1 data

TMP

All data is constant because the compiler can not deal with dynamic data. One of the goals of TMP is to hand as many computations as possible to the compiler. The compiler can handle data only when it is constant. Constness can be thought of as an effect of the absence of an assignment operator. This lack of assignment leads to programs that appear to be very functional.

Compile-time constant variables may be used as template variables because they are constant at the time of program compilation and therefore during template processing. The most common instances of compile-time

constants are

- any literal number or character in the source code
- `const` global variables
- `const` local variables initialized to other compile-time constants
- `static constant` structure data members. Static structure data members must be defined to compile-time constants or otherwise link errors will arise after compilation.

Run-time constants are constant variables initialized to non-constant variables or other run-time constants. They are defined during run time but cannot be changed once defined. The most common instances of run-time constants are:

- `const` function parameters
- `const` non-static data structure variables

In common C++ programming it is not necessary to differentiate between compile- and run-time constants. When using TMP however, there is a big difference between these two types. This is because all TMP computations must be carried out by the compiler before any run-time computations occur. Because of this, run-time constants can not be used for TMP.

Non-constant variables make up the rest of the class of data available in C++. They are unusable in TMP.

Post Template Code (PTC) refers to all code that does not depend on template processing. All C++ code is eventually reduced to PTC after all templates have been instantiated.

PTC

data is either:

- compile-time constant
- run-time constant
- non-constant

2.2 execution

TMP

All execution is performed at compile time through determination of type signatures for template instantiations.

PTC

Some compiler optimizations allow simple parts of the program to be computed at compile time, but most computations are performed at runtime.

2.3 pattern matching

TMP

Pattern matching is available with template specialization. It is used extensively in the fibonacci example. When template specialization is viewed from the perspective of the hijacking table, we can make the following observation:

Template specialization lets the programmer associate the definition of a function with the type of the data the function is called with.

PTC

Pattern matching is not available as a language feature.

2.4 dominant paradigm

TMP

Functional. As a direct result of the absence of state changes, the only available paradigm to operate in is declarative, with computations being carried out by recursive function calls.

PTC

Procedural, arguably OO. Some libraries allow for functional programming as well (e.g. FACT, Funk, etc..).

3 The Funk functional programming system

In this section, the Funk Functional syntax and programming framework for C++[Hal00] is outlined. Please note that many of the definitions used in this section are re-stated to reflect new concepts being introduced.

3.1 expression templates

Expression templates (ETs) are template instantiations that represent recursively constructed expressions. All Funk code is based around evaluating or aggregating other expression templates. All ETs are formed around atomic ETs such as ET variables or ETs that contain C++ function pointers. ET variables are defined by instantiations of the struct template `ETvar`. An `ETvar` needs two parameters for instantiation: a

name variable to differentiate it from other variables in the same scope, and a type variable that specifies the type of the data it holds. The name variable is just a character template parameter. Unfortunately this limits Funk variable names to single letters only. Future versions of Funk may allow for more descriptive variable names.

```
template<char n, class T>
struct ETvar { };
```

Using the above definition of ETvar, we can define some variables for use in constructing Funk expression templates. The following code defines ETvars called a, b, c, and d.

```
ETvar<'a',int> a;
ETvar<'b',int> b;
ETvar<'c',int> c;
ETvar<'d',int> d;
```

ETs can also be defined to create algebraic expressions. For example, we can define an ET plus that combines two other ETs:

```
template<class LHT, class RHT>
struct plus { };
```

```
template<class LHT, class RHT>
struct times { };
```

where LHT and RHT are expression templates.

The following table shows algebraic expressions and their corresponding expression template typenames which are composed of the previous definitions and ET variables.

| | |
|----------------------|--|
| algebraic expression | expression template |
| $a + b$ | <code>plus<ETvar<'a',int>,ETvar<'b',int> ></code> |
| $(a + b) * (c + d)$ | <code>times<plus<ETvar<'a',int>,ETvar<'b',int>>,plus<ETvar<'c',int>, ETvar<'d',int>>></code> |

3.2 lambda and apply type system

Partial application of functions is a nice feature for any functional programming system. Funk implements this particular feature using the types `lambda` and `apply` and several utility metaprograms. As their names suggest, `apply` is used to hold the values of arguments to which functions have been applied, whereas `lambda` is used to specify the need for and type of a function's parameter. This subsection deals exclusively with the C++ types used to describe of Funk lambda expressions. The mechanisms used to apply data to and evaluate Funk lambda expressions is covered in the section on evaluating Funk expression templates.

Initially, we can begin this discussion with trivial definitions of `lambda` and `apply`:

```
template<class V, class ET> lambda { };
```

Any Funk lambda expression that has type `lambda<A,ET>` corresponds to the lambda expression $\lambda a.et$, where a has type `A` and et has type `ET`.

```
template<class V, class ET> apply { };
```

Expression templates can be turned into lambda expressions by embedding them into a series of instantiations of the `lambda` template. For example, the `ET`:

```
plus< ETvar<'a',int>,ETvar<'b',int> >
```

can be turned into a `lambda` expression like this:

```

lambda< ETvar<'a',int>,
    lambda< ETvar<'b',int>,
        plus< ETvar<'a',int>,
            ETvar<'b',int> > > >

```

where `lambda`'s first template parameter holds information about the variable that it manages. The expression template representing the body of the lambda expression is the second template parameter for the `lambda` type.

The above structure represents the following lambda expression:

$$\lambda a.(\lambda b.(a + b))$$

When a Funk lambda expression is partially applied to an argument, the type of the resultant application is `apply` instantiated with the same arguments the former `lambda` was instantiated with. So if a lambda expression having the type of the above instantiation were applied to an argument, its type would become:

```

apply< ETvar<'a',int>, // note the apply instead of the lambda
    lambda< ETvar<'b',int>,
        plus< ETvar<'a',int>,
            ETvar<'b',int> > > >

```

and after application to another argument, its type transiently becomes:

```

apply< ETvar<'a',int>, // note the apply instead of the lambda
    apply< ETvar<'b',int>,
        plus< ETvar<'a',int>,
            ETvar<'b',int> > > >

```

But because the type of this lambda expression is a fully applied lambda expression, it must be reduced to

the lambda expression's expression template's final type in order to be usable for further computation. The type of this expression template is therefore simply `int`.

The mechanisms that perform type translations of Funk lambda expressions into their applied state are discussed in the following section on evaluating Funk expression templates.

3.3 evaluating Funk expression templates

The Funk framework uses TMP techniques to build complex expression templates and determine the resultant type of lambda expressions. So while it is possible to perform computations with template metaprogramming, their role in Funk is primarily to support it as a compile-time type-safe sub-language of C++ — not to perform fancy computations like the Fibonacci example. This role for TMP in Funk is very necessary, however, because there is no other way in C++ to compute the types necessary for determining final types for lambda expression application at compile-time.

3.3.1 dealing with data

Once the compiler instantiates all TMPs, the execution of programs written with Funk is the same as the execution of any other C++ program; program input and output happen during runtime. All the features of Funk — expression templates, lambda expressions, function composition, and partial application, etc... — only put a pretty context around otherwise standard C++ operations. The bottom line is that the results produced by Funk programs are computed by the same mechanisms that produce the results of any other C++ program. In order to use these mechanisms, Funk must be able to work with data. Until now, the Funk templates that we have defined have not been designed to hold any data members. Let us now redefine the necessary templates to be able to work with data. We begin with the `apply` template by adding a data member that holds the value of the data lambda expressions are applied.

```
template<class V, class ET>
```

```

struct apply {
    apply(V _v, ET _et) : v(_v), et(_et) { }

    V    v; // value of lambda expression parameter

    ET   et; // value of lambda expression body
};

```

So now, because any ET might contain an `apply`, any aggregating ET must be able to store the run-time values of its subexpressions. Expression templates can be modified to store the values of their ET template parameters by adding member variables that have the type of their subexpressions. Redefining all remaining relevant Funk example templates to cope with data, we get:

```

template<class LHT, class RHT>
struct plus {
    plus(LHT _lhs, RHT _rhs) : lhs(_lhs), rhs(_rhs) { }

    LHT    lhs; // value of left hand side ET

    RHT    rhs; // value of right hand side ET
};

```

```

template<class LHT, class RHT>
struct times {
    times(LHT _lhs, RHT _rhs) : lhs(_lhs), rhs(_rhs) { }

    LHT    lhs;

    RHT    rhs;
};

```

```

template<class V, class ET>
struct lambda {
    lambda(V _v, ET _et) : et(_et) { }

    ET    et;
};

```

```

};

template<char n, class T>
struct ETvar {
    static T value; // static data member associates
                    // 'value' with the ETvar<n,T> type.
};

```

Because a variable's identifier is defined as part of its type, so must be the variable's value. Declaring data members as static lets us do just that. This is why the `value` data member for `ETvar` is declared static.

3.3.2 defining an expression template's *final type*

When a lambda expression is applied to its final argument, the type of the application can no longer be a series of nested `applies`, but must reflect the *final type* of the lambda expression's ET. The final type of an expression is the result type of the expression's outermost operator. When declaring final types for Funk's predefined mathematical operators, it sufficed to base their final type's on the final type of the ET's leftmost operand. This allowed for simple type signature definitions for these operators. For example, the final type of the expression template `plus<ETvar<'a',int>,ETvar<'b',int> >` is the final type of `ETvar<'a',int>`, so if `ETvar<'a',int>`'s final type is `int`, `plus<ETvar<'a',int>,ETvar<'b',int> >`'s final type is also `int`.

To define the final type for any ET, define a specialization to the structure template `final` (defined below) that describes the ET's final type. As an example for `plus`, `final` should be specialized as follows:

```

template<class>
struct final { };

template<class LHT, class RHT>

```

```

struct final<plus<LHT, RHT> > {
    typedef final<LHT>::type type; // return the type of the first argument
}

```

specializing `final` is similar but not identical to defining polymorphic type signatures in a language like Haskell. The above `final` type specialization could be thought of as declaring the following type signature:

```
plus::a->b->a
```

3.3.3 switching the type of fully applied lambda expressions

The job of switching the type of the final application of a Funk lambda expression to the expression template's final type is handled by a TMP called `value::type`. This TMP is used by the lambda expression application function (defined in “applying values to Funk lambda expressions”) every time a `lambda` expression template is applied to an argument.

3.3.4 final types of expression templates

When a lambda expression is applied to all its arguments the type of the expression must be turned into the result type of the lambda expression. Switching the type of the expression is the job of a TMP called `value::type`. `value::type` takes a lambda expression that has just had its outermost lambda instantiation replaced with an `apply` and returns a type derived from the following logic: If given template parameter is made entirely of `apply` instantiations, then it is ready to to be transformed into it's body expression's final type, so return the lambda expression's body expression template, otherwise, return the entire unchanged lambda expression. In the following definition, `value::type` is actually just a convenience metaprogram that feeds the same type to both template parameters of a TMP called `value_::type`, which does the real work.

```

template<class E0, class T> class value_
    { typedef final_type<T>::type type; }; // this is an important default
                                           // case because it gets instantiated
                                           // for any parameter that is not an
                                           // apply<V,ET> or lambda<V,ET>.

template<class E0, class V, class ET>
class value_<E0, apply<V,ET> >
    {typedef value_<E0, ET>::type type;}; // drill into the instantiations
                                           // to find the expression template.

template<class E0, class V, class ET>
class value_<E0, lambda<V,ET> >
    {typedef E0 type;};

```

If the last specialization in the above definition is instantiated with a lambda, then ET is still a partially applied lambda expression and the expression should not be collapsed into the final type of the expression template². Continuing,

```

template<class ET>
class value
    { typedef value_<ET,ET>::type type;};

```

`value::type` feeds the same type expression to both parameters of `value_::type` (note the underscore.)

`value::type` can also be used to compute the result type for an evaluation function. `eval` is a C++ function template that produces the tangible result for any expression template evaluation and is the last transformation that an ET goes through before it is exposed to any non-Funk operations in regular C++.

²This is similar to Haskell's method of handling evaluation of functions that have not been applied to their complete set of arguments; Haskell will just collect arguments until every parameter has a corresponding argument, at which point the expression gets re-written based on the collected arguments.

Below, `eval` is defined for `ETvars`, `lambdas`, and `apply` ETs:

```
template<class V,class ET>
value< apply<V,ET> >::type
eval(apply<V,ET> a) {
    V::set_value(a.v);
    return value< apply<V,ET> >::type(eval(a.et));
}
```

```
template<class V,class ET>
lambda<V,ET> eval(lambda<V,ET> l)
    {return l;}

```

```
template<class V, class T>
T eval(ETvar<V, T>)
    { return ETvar<V, T>::value; }
```

```
template<class LHS, class RHS>
result<plus<LHS,RHS> >::type eval(plus<LHS,RHS> p)
    {return p.lhs + p.rhs;}

```

The `eval` overload defined for `applies` sets `V`'s `static` member `data` to the value stored in `a.v`. Because it is `static`, `V::data` is accessible in every instance of `V`. This allows the `data` value for `V` to be accessed wherever `V` is accessible. Finally, `eval<apply>` returns `eval(a.et)`.

3.3.5 applying values to lambda expressions

The TMPs and functions are now properly defined to allow us to define a function that can handle the application of a lambda expression to an argument. It will take two arguments; one a lambda expression, and the other the argument that the lambda expression is to be applied to, and return the proper lambda expression or result type that the application results in.

```
template<char n, class A, class ET>
result<apply<ETvar<n,A>,ET> > apply_lambda_to_arg(lambda<ETvar<n,A>,ET> l, A a) {
    eval(apply<ETvar<n,A>,ET>(a,l.et));
}
```

This function does not really do much work in itself, it merely creates an instance of `apply` based on the lambda provided as a parameter feeds the instantiation to `eval`.

3.4 an example Funk evaluation

Using the above definitions, it is now possible to apply arguments to lambda expressions. Follow this example to get a good grasp on how the process works.

evaluate: $((\lambda a.(\lambda b.a + b))3)4$

in C++, the above expression would be represented by:

```
apply_lambda_to_arg(
    apply_lambda_to_arg(
        lambda< ETvar<'a',int>,
            lambda< ETvar<'b',int>,
                plus<
                    ETvar<'a',int>,
                ETvar<'b',int> > > >(),
            3),
    4)
```

Although C++ does not implement substitution for evaluation of its expressions, it is the best way to express the evaluation of this Funk lambda expression. The following trace is formatted with the expression evaluation followed by an '&' followed by any relevant data values. Evaluation steps are separated with the token '==='.

```

apply_lambda_to_arg(
  apply_lambda_to_arg(
    lambda< ETvar<'a',int>,
      lambda< ETvar<'b',int>,
        plus<
          ETvar<'a',int>,
          ETvar<'b',int> > > >(),
        3),
    4)
&
===

apply_lambda_to_arg(
  eval( apply< ETvar<'a',int>,
    lambda< ETvar<'b',int>,
      plus<
        ETvar<'a',int>,
        ETvar<'b',int> > > >()),
    4)
&
===

apply_lambda_to_arg(
  value<apply< ETvar<'a',int>,
    lambda< ETvar<'b',int>,
      plus<
        ETvar<'a',int>,
        ETvar<'b',int> > > >::type(),
    4)
&
ETvar<'a',int>::value == 3
===

apply_lambda_to_arg(
  value_<apply< ETvar<'a',int>,
    lambda< ETvar<'b',int>,
      plus<
        ETvar<'a',int>,

```



```
ETvar<'a',int>::value == 3
```

```
===
```

```
value< apply< ETvar<'a',int>,
            apply< ETvar<'b',int>,
                plus<
                    ETvar<'a',int>,
                    ETvar<'b',int> > > >::type(
                        eval( apply< ETvar<'b',int>,
                                plus<
                                    ETvar<'a',int>,
                                    ETvar<'b',int> > >))
```

```
&
```

```
ETvar<'a',int>::value == 3
```

```
===
```

```
value< apply< ETvar<'a',int>,
            apply< ETvar<'b',int>,
                plus<
                    ETvar<'a',int>,
                    ETvar<'b',int> > > >::type(
                        value<plus<
                            ETvar<'a',int>,
                            ETvar<'b',int> > >::type(
                                eval( plus<
                                    ETvar<'a',int>,
                                    ETvar<'b',int> > >)))
```

```
&
```

```
ETvar<'a',int>::value == 3
```

```
ETvar<'a',int>::value == 4
```

```
===
```

```
value< apply< ETvar<'a',int>,
            apply< ETvar<'b',int>,
                plus<
                    ETvar<'a',int>,
                    ETvar<'b',int> > > >::type(
                        value<plus<
                            ETvar<'a',int>,
                            ETvar<'b',int> > >::type( 7 ) )
```

```
&
```

```
ETvar<'a',int>::value == 3
```

```
ETvar<'a',int>::value == 4
```

===

```
value< apply< ETvar<'a',int>,
          apply< ETvar<'b',int>,
                plus<
                  ETvar<'a',int>,
                  ETvar<'b',int> > > >::type(
                    value_< plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> >,
                    plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > >::type( 7 ) )
```

&

```
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4
```

===

```
value< apply< ETvar<'a',int>,
          apply< ETvar<'b',int>,
                plus<
                  ETvar<'a',int>,
                  ETvar<'b',int> > > >::type(
                    final_type< plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > >::type( 7 ) )
```

&

```
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4
```

===

```
value< apply< ETvar<'a',int>,
          apply< ETvar<'b',int>,
                plus<
                  ETvar<'a',int>,
                  ETvar<'b',int> > > >::type(
                    final_type< ETvar<'a',int> >::type( 7 ) )
```

&

```
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4
```

===

```
value< apply< ETvar<'a',int>,
          apply< ETvar<'b',int>,
                plus<
```

```

        ETvar<'a',int>,
        ETvar<'b',int> > > >::type(int( 7 ) )

&

ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

value_< apply< ETvar<'a',int>,
              apply< ETvar<'b',int>,
                    plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > >,
              apply< ETvar<'a',int>,
                    apply< ETvar<'b',int>,
                          plus<
                            ETvar<'a',int>,
                            ETvar<'b',int> > > >::type(int( 7 ) )

&

ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

value_< apply< ETvar<'a',int>,
              apply< ETvar<'b',int>,
                    plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > >,
              apply< ETvar<'b',int>,
                    plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > > >::type(int( 7 ) )

&

ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

value_< apply< ETvar<'a',int>,
              apply< ETvar<'b',int>,
                    plus<
                      ETvar<'a',int>,
                      ETvar<'b',int> > >,
              plus<
                ETvar<'a',int>,
                ETvar<'b',int> > > >::type(int( 7 ) )

```

```

&
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

final< plus<
    ETvar<'a',int>,
    ETvar<'b',int> > >::type(int( 7 ) )

&
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

final< ETvar<'a',int> >::type(int( 7 ) )

&
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

int(int( 7 ) )

&
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

===

7

&
ETvar<'a',int>::value == 3
ETvar<'a',int>::value == 4

```

3.5 building funk structures with function templates

The automatic type inferencing features of function and operator templates allow us to create expression templates without having to explicitly instantiate structure templates. For example, we could define a function template `make_et_plus` that takes two expression templates as arguments and returns an instance

of `plus<T,T>` whose template parameters are the arguments provided to the function. So, we are in effect using the types automatically determined by the C++ compiler during instantiating of the function template to automatically instantiate a our own expression template with the proper types.

```
template<class T1, class T2>
plus<T1,T2> make_et_plus(T1 lhs, T2 rhs) {
    return plus<T1,T2>(lhs,rhs);
}
```

It is not necessary to provide the type for a template argument when calling a function template if the argument's type can be deduced by the C++ compiler from the parameters passed to the function. So, assuming that we have ET variables `a` and `b` defined as before, we can create instances of the type:

```
plus<ETvar<'a',int>,ETvar<'b',int> >
```

using our function template simply by writing the following C++ code:

```
make_et_plus(a,b)
```

without the need to state any typenames.

3.5.1 using type resurrection and overloaded operators to to build expression templates

Creating instances of expression templates can be further simplified through the use of operator overloading. For our example, we would like to overload the operator `+` so that it can take two expression templates and return a `plus` aggregate of the two argument types. However, this brings up a small problem: we need to enable the `plus` operator to operate on all types of expression templates. Simply overloading the `plus` operator with a template that has two type variables for the LHS and RHS would ruin overloading the `plus` operator for any other types because our overload would be too general and therefore match all calls to operator `+`. So to be a good namespace citizen, the `plus` operator should only be overloaded for expression templates. Restricting operator overloads to certain classes of types can be accomplished by using object-oriented techniques to group all expression templates under one basic superclass and only overloading for the

superclass. Doing this brings up another small problem. When the superclass type is matched as a function parameter, it is not possible to recover the original type of the parameter, but is necessary to recover the original type because the type *is* the expression. For example,

```
struct ET { } // superclass expression template

plus<ET,ET>
operator + (ET lhs, ET rhs) {
    return plus<ET,ET>(lhs,rhs);
}
```

`lhs` and `rhs` might have been variables or more complex expression templates but have now been sliced³ to their superclass type and all information about what they actually were has been truncated. As it is, this approach does not work but can be solved using a system I call *type resurrection*. A type resurrection system can be set up by making the superclass actually a structure template with one template parameter. All expression templates then should inherit from an this superclass instantiated with the type of the expression template. Now when the type of an expression template is sliced upon being passed as an argument to a restrictive function template, it's original type can be resurrected from the template parameter of the sliced object.

Let us now redefine the necessary parts of our system to allow for this. We start by defining a base structure template `ET<T>`:

```
template<class>
struct ET { };
```

Now we need to redefine our definitions of `ETvar` and `plus` so they derive themselves from `ET<T>`:

```
template<char n, class T>
struct ETvar : ET<ETvar> // derive from ET<ETvar<n,T> > for type resurrection
{
    static T value;
};

template<class LHT, class RHT>
```

³slicing occurs when data members exclusive to a subclass get truncated as an object instance is cast as its superclass

```

struct plus : ET<plus<LHT, RHT> >    // derive from ET<plus<LHT, RHT> > for type resurrection
{
    plus(LHT _lhs, RHT _rhs) : lhs(_lhs), rhs(_rhs) { }
    // this is just a convenience constructor.
    LHT    lhs;
    RHT    rhs;
};

```

It is now possible to overload the plus operator for expression templates. But along with preserving type information it is also necessary to preserve data information that parameters contain for reasons outlined in the discussion of the `apply` expression template. To preserve data information as well as type information but still have the class match as a superclass, an argument type of `ET<T>&` must be used for the parameters of function and operator templates that will resurrect types. When an argument is a reference, slicing only affects the object's type and not its data. This allows us to safely cast the argument back to its original type and still retain its data integrity.

```

template<class E1, class E2>
plus<E1,E2> operator + (ET<E1>& lhs, ET<E2>& rhs) {
    return plus<E1,E2>(
        static_cast<E1&>(lhs),    // inflate to original type by using static_cast<T&>
        static_cast<E2&>(rhs));
}

```

Using the above operator definition and our set of variable definitions, we can write things like $a + b$ and the result will be an expression template.

Similar definitions can be used to define other expression template building operators such as the rest of the common algebraic operators and functional operators for partial application and function composition.

3.6 syntax

One of the goals of Funk is to become a self contained sublanguage of C++. Funk also tries to imitate functional conventions as much as possible.

3.6.1 lambda expressions

Lambda expressions are created by making a comma separated list of parameters followed by the `--` and `>` operators then followed by the expression template representing the body of the Funk lambda expression. The comma separated list is actually a series of invocations of operator `,`. The operator `--` is simply eye candy, but makes for a more comprehensible syntax. operator `>` converts the list created by the comma operator and an expression template into an actual Funk lambda expression. This creative use of C++ operators leads to a very Haskell-like syntax for creating lambda expressions. The following code, although it may not look it, is actually legal C++ code once the Funk libraries are loaded.

```
x-->(x*x);  
(a,b,c)-->((a+b)/c);
```

3.6.2 partial application

Functions are partially applied arguments through the use of operator `<<`. For example:

```
((x,y)-->((x+y)/(x*y))) << 3 << 4;
```

3.7 shortcomings of Funk as a sublanguage of C++

3.7.1 polymorphism

Funk currently does not offer support for polymorphic types in expression templates. This rules out many of the uses of higher order functions. Future versions may incorporate this functionality.

3.7.2 naming Funk objects

Funk does provides some nice functional features to the C++ programmer, but it also has some limitations imposed by the C++ language that cannot be circumvented. It is not possible to name an expression template without stating its entire type in the declaration. It is *possible* to state an expression template's type, but

doing so is usually not worth the while of the programmer because typenames for ETs get very complicated very fast. One solution to this problem is to incorporate Funk into yet another level of meta-language that when compiled, generates one conglomerate statement that includes a series of instantiations of Funk objects, effectively leaving all type inferencing on the compiler. The objects could then be referenced through a compile-time system using TMP techniques, thereby eliminating any run-time overhead. A language that attempts to implement this scheme is currently under development and will be presented in a future report.

References

- [Hal00] Thomas Hallock. Funk: A c++ library that enables light functional programming. Technical report, Available: <http://wnt.cc.utexas.edu/~antialias/funk/>, 2000.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language Third Edition*. Addison-Wesley, 1997.
- [Vel02] Todd Veldhuizen. C++ templates as partial evaluation. Technical report, Available: <http://osl.iu.edu/~tveldhui/papers/pepm99/>. Pervasive Technology Labs, 2002.