Functional programming with C++ Templates

Andreas Molzer

Sunday 28th May, 2017

Abstract

In this paper the usability of templates for functional programming techniques is explored. First, a notation for functional structures within the compile time construct of `C++` is established. Then, an implementation of several basic and useful features for general purpose programming in a functional language are provided. These include some frequently used higher-order functions as well as a simple type for representing lists. The applicability of the generic constructs is shown by implementing the list monad within the set programming paradigm.

Contents

1 Motivation

It has been noted on many occasions since the introductions of templates in C++98 that the compile time features offered make them a Turing complete language in itself. This means that a programmer can, with enough effort, convert any algorithm into a form which uses Templates and which any standard compliant compiler is able to evaluate.

It does, however, not imply that the resulting code is readable nor understandable after this transformation. In fact, some programming languages were invented with the particular idea of having overly complex, source code while still maintaining Turing-Completeness [Malbolge, ThatFunctionalOne]. For this reason, the feature remained little more than a neat fact for computational scientists and an inside joke for those with deeper understanding of the C++ language.

This paper is an effort to change this conception and introduce notation and code standards in order to turn this subfeature into a powerful tool for computations at compile time. Imaginable use cases include the following:

- Signing compilation units
- Replacement of parser generators such as YACC[4]
- Randomizing memory layouts at compile time

We will discuss some of those tools at the end of this paper.

Firstly, we embed the structures of functional programming into syntax constructs available in C++. Several approaches are compared and we choose the one which incorporates as many common template meta programming techniques as possible.

2 Functional embedding

We will first discuss the core of our implementation, that is the necessary types and operators to provide a functional foundation. These are then expanded on, to form some core structures as well as syntactical constructs to ensure programming can be done in a natural, readable syntax. The language will then be classified and its boundaries explored.

The `C++` code discussed here is based on working draft n4659 [7] for `C++1z`, implemented in the clang compiler [1].

2.1 Functional languages

A functional language consists of three primary structures or operations.

Variables Variables might refer to any basic datatype or any function but there is no notational difference between the two. Variables can be used as arguments for any other function and be evaluated as long as the replacement of the function body by the value yields a valid expression. For example, the identity function can be expressed regardless of argument type as follows

$\lambda x : x$

Function definition Every function in lambda calculus names exactly a single argument and a function body, in the form

$\lambda x : expr$

where $expr$ is any expression which may or may not involve the named argument x.

Function application Function applications is notated as

$x\ y$

applying the function x to the argument x. In order to evaluate this expression, we replace every occurrence of the argument of function x in the body of x with the value of y.

$(\lambda x : x)\ 3 \equiv (x)[x/3] \equiv 3$

2.2 Functional structures and Templates

The next goal is to accurately replicate all structures from the previous section with C++. We must first answer the question of how to notate variables.

Variables It is straightforward to use identifier to refer to them, as it is already possible to refer to template parameters by name. There are potentially multiple ways with which to denote a variable though, as many as there are different template parameter types. Using integral types has obvious short comings such as not being able to enforce any real type restrictions. Template typename arguments leave enough room to accommodate all parts needed for our purpose and while it was theoretically possible to use template template parameters instead, it would lead to absurd syntax constructs and unnecessary clutter. Basic symbols can now be named simply by declaring a struct.

```
1 struct Zero;
```

If we want to build one symbol from one or several others, we can do so by declaring template parameters. The declaration can then be used similar to constructors in other languages.

```
1 template<typename Predecessor> struct Number;
2 using One = Number<Zero>; // example usage
```

As we will see later, we can even semi-automatically build functions for these constructors.

Function definition As we noted before, each function is referred to by a struct definition or declaration of its own, just like variables and structures. The real definition happens inside the function, where a templated alias declaration (or member struct) is defined, naming exactly a single template type argument and which will, when provided with that argument, resolve to the type identified by replacing the occurences of its name with the given value.

This allows a rather simplistic implementation of the identity function.

```
struct id {
  template<typename Arg> using expr = Arg;
};
```

At the same time, we can implement `const_` (which ignores its second argument) in a similar fashion.

```
template<typename Val> struct Const {
        template<typename Arg> using expr = Val;
};
struct const_ {
  template<typename Arg> using expr = Const<Arg>;
};
```

Using a member alias declaration or a member struct allows one to combine types and functions via inheritance.

Function application Based on the concept introduced for function definitions, it is straightforward to implement function application. We simply access the contained member templated alias declaration, provide the argument and declare it as the result.

```
template<typename F, typename Arg>
using ApplyArgument = typename F::template expr<Arg>;
```

2.3 Categorization and Limitations

Every argument given in the form of a `typename` is completely evaluated by the compiler before it is inserted in the place of a template parameter. This enforces a strict evaluation of all data. As is later shown by definition and by example, it does however not restrict recursion capability of this language. In fact, partially because it interfaces nicely with other meta programming techniques, it is not too hard to show that it is Turing complete.

3 Library functionality

Although the aforementioned notations give us a way of formalizing the mechanisms of functional programming with the C++ language, we are still far from

the goal of usability. Many meta template programming techniques will not conform with the requirements we set. Additionally, we have restricted ourselves to a very narrow definition for every feature.

The need for converters and other syntactic sugar will be a focus in this chapter. In order to show actual applicability of the devised constructs, an implementation of a basic list type is exercised. Similar structures have already been shown to be useful in meta programming with templates [2] but the results presented here far exceed the capabilities of a pure structural type. The functional approach allows not only for implementing common operations but actually promotes the list type to a full monad, able to operate on extensive mathematical constructs.

3.1 Standard import - `TypeFunction`

Let us first tackle the biggest compatibility issue at hand, that is implement ways of importing standard library meta programming structures into our programs. We can then move on to templated type declarations.

Several functions for transforming types exist in the standard library. These are classes or structs with one or more template arguments which declare a member alias `type` naming the result of their operation. For example `std::add_pointer`

```
namespace std {
  template<typename T> struct add_pointer {
    using type = T*;
  };
}
```

Firstly, we need to be able to deduce the number of template arguments. Restricting the number to an arbitrarily chosen upper bound, this can be done fairly easily. We use function overloading and type deduction in order to choose from a linear number of templated functions.

```
template<template<typename>typename>
auto _deduct_function() -> ::std::integral_constant<size_t, 1>;
template<template<typename,typename>typename>
auto _deduct_function() -> ::std::integral_constant<size_t, 2>;
/* Several more declarations for other parameter counts */
template<template<typename...> typename Arg>
constexpr const static size_t args_count =
  decltype(_deduct_function<Arg>())::value;
```

Code listing 1: Deducting the number of template parameters

The templated variable `args_count` gives us the number of arguments of a template declaration, structs as well as aliases, in the form of a constexpr variable declaration.

This enables us to provide a declaration for converting any standard meta templating function operating purely on types into one usable by the functional implementation.

```
template<template<typename...> typename U,
         size_t nargs = helper::args_count<U>,
         typename... AppliedArguments>
struct PartialTypeFunction;

// Overload for n-1 arguments, next argument completes the call
template<template<typename...> typename F, typename...Args>
struct PartialTypeFunction<F, sizeof...(Args)+1, Args...> {
        template<typename LastArg>
        using expr = typename F<Args..., LastArg>::type;
};
// All other cases
template<template<typename...> typename F, size_t nargs,
         typename... Args>
struct PartialTypeFunction {
        template<typename NextArg>
        using expr = PartialTemplateFunction<
            F, nargs, Args..., NextArg>;
};
template<template<typename...> typename F> using TypeFunction =
  PartialTypeFunction<F>;
```

Code listing 2: Partial function application for standard functions

As we can see, each specialization of `PartialTypeFunction` has the form discussed in 2.2. Each function application will add an additional argument, and then evaluate to `U<Args...>::type`, when all arguments were provided. `TypeFunction` refers to the partial application without any arguments.

3.2 Common functions

The following functions could be implemented by any user of the library without bigger problems but they are so common, and their definitions very elementary. In this way, they serve as great examples of expressiveness and opportunities for composition. Their implementation make great use of `TypeFunction` in order to ease the definition of future functions, as can be seen later in the implementation of list monads.

Variadic Apply Often times, a function is to be applied to multiple arguments at once. In such cases it would be tedious to explicitly write down all applications in code. Instead, variadic templates make it possible to apply an arbitrary amount of arguments in order, similar Haskell or other functional languages.

```
template<typename F, typename... args>
struct _apply_all;
template<typename F>
struct _apply_all<F> {
  using result = F;
};
template<typename F, typename FirstArg, typename... args>
struct _apply_all<F, FirstArg, args...> {
  using _applied = ApplyArgument<F, FirstArg>;
  using result = typename _apply_all<_applied, args...>::result;
};

template<typename F, typename ...args>
using Apply = typename _apply_all<F, args...>::result;
```

Code listing 3: Applying several arguments after one another

Function Composition Implements mathematical function composition ∘. This higher-order function takes two functions and returns a functions which when applied to an argument `Arg` first applies the second function to `Arg` and then applies the first function to the result.

Compose

Signature: `(B -> C) -> (A -> B) -> (A -> C)`
Definition: `compose g f`
Effect: Constructs a function representing `g f`, i.e. first applies its arguments to `f` then applies `g` to the result.

```
template<typename F, typename G, typename Arg>
struct _compose {
  using type = Apply<F, Apply<G, Arg>>;
};
using compose = TypeFunction<_compose>;
```

Code listing 4: Function composition

The Flip function This is probably one of the simplest and most used functions. It reverses the order of the first two parameters of any function.

Flip

Signature: `(A -> B -> C) -> (B -> A -> C)`
Definition: `flip f`
Effect: The resulting function reverses the order of its first two arguments before applying `f` to them.

```
template<typename F, typename B, typename A>
struct _flip {
  using type = Apply<F, A, B>;
};
using flip = TypeFunction<_flip>;
```

Code listing 5: Implementation of the common flip function

3.3 The list type

Any good standard library provides a solid ground work of reusable types as well as structures for efficient implementation of user defined types. The following sections showcase some important functional examples rooted in the definitions above, used to implement a fundamental list type. It also explores the concept of monads.

Definition While the elementary type definitions follows very closely previously published implementations [2], it is nevertheless specified in its entirety for the sake of completeness. The interface of all related functions follow the concepts provided in the Haskell Standard Library [5]. This provides programmers experienced in functional programming with an easily recognizable style and at the same time ensures a practically tested and logically consistent design. We begin with the basic definitions, the two constructors of our list type:

```
// List a = Nil | Cons a (List a)
struct Nil;
template<typename head, typename tail>
struct Cons;
```

Code listing 6: The two constructors of a list

There are two possible options for constructing a list. The first is for it to be empty while the second is a prepending an element to the front of an already existing list. While no type safety is enforced via the above declarations, such unsafe constructions by a user will simply be invalid as arguments to the following functions, failing to evaluate.
After these definitions, we can code simple introspection tools, retrieving information about any given list. The most simple of these are `head` and `tail`, which return from a list the first element, everything but the first element respectively.

```
template<typename L> struct Head;
using head = TypeFunction<Head>;

template<typename head, typename tail>
struct Head<Cons<head, tail>> {
```

```
  using type = head;
};

template<typename L> struct Tail;
using tail = TypeFunction<Tail>;

template<typename head, typename tail>
struct Tail<Cons<head, tail>> {
  using type = tail;
};
```

Code listing 7: Retrieving the head and tail of a list

Additional list functions In order to complete our goal of a fully functional monadic implementation of list, we need three additional functions. These are concatenation, `concat`, the ability to apply a function to each element of a list, `map`, and the ability to reduce a list with a binary operator, `fold`. These are all described in the following paragraphs. Signatures are denoted via typed notation, but this is solely to increase readability.

Concat
Signature: `List -> List -> List`
Definition: `concat first second`
Effect: The resulting list will contain all element of `first` followed by the elements in `second`, in the order they appear in these respective lists.

Map
Signature: `(A -> B) -> List A -> List B`
Definition: `map operator list`
Effect: Replaces every element `e` in `list` by the element `operator e`, the application of `e` to the unary `operator`. This is a higher-order function.

Fold
Signature: `(I -> A -> I) -> I -> List A -> I`
Definition: `fold operator initial list`
Effect: Fold the `operator` over the `list` using `initial` as a starting elements. Imperatively, this could be described as splitting the list into `head` and `tail`, then calculating a new initial element `initial_` as `operator initial head`, and recursively calling `fold operator initial_ tail`.

```
// All of these functions are implemented via specialization
template<typename L, typename J> struct Concat;
using concat = TypeFunction<Concat>;
template<typename F, typename I, typename L> struct Fold;
using fold   = TypeFunction<Fold>;
template<typename F, typename L> struct Map;
using map    = TypeFunction<Map>;

```

```
template<typename L>
struct Concat<Nil, L> { // Concat [] L
  using type = L;
};
template<typename head, typename tail, typename L>
struct Concat<Cons<head, tail>, L> { // Concat (head:tail) L
  using type = Cons<head, Apply<concat, tail, L>>;
};

template<typename F, typename I>
struct Fold<F, I, Nil> { // Fold F I []
  using type = I;
};
template<typename F, typename I, typename head, typename tail>
struct Fold<F, I, Cons<head, tail>> { // Fold F I (head:tail)
  using type = Apply<fold, F, Apply<F, I, head>, tail>;
};

template<typename F>
struct Map<F, Nil> { // Map F []
  using type = Nil;
};
template<typename F, typename head, typename tail>
struct Map<F, Cons<head, tail>> { // Map F (head:tail)
  using type = Cons<Apply<F, head>, Apply<map, F, tail>>;
};
```

Code listing 8: Implementation of several list related functions

3.4 List as a monad

In the interpretation of functional programming, a monad consists of five essential functions, `return`, `bind`, `fmap`, `join` and the `kleisli` operator. Only a subset of those is needed in order to fully characterize a monad. By defining `fmap`, `return` and `join`, it is for example possible to recover definitions for the other missing functions [6, 3].
The monadic interpretation of lists allows for a computation operating on several inputs, single instruction multiple data so to speak, where each function application might yield no, one or multiple results. The result is a concatenated list of the individual results [6].

```
using list_fmap = map;
struct list_return {
  template<typename T> using expr = Cons<T, Nil>;
};
```

```
5 using list_join = Apply<fold, concat, Nil>;
```

Code listing 9: Basic monadic functions of list

Now all that is necessary for the complete definition of the list monad are its operations `bind` and `kleisli` which can be composed from the previous operations.

```
1 using list_bind = Apply<compose, Apply<compose, list_join>,
2                                  Apply<flip, map>>;
3 using list_kleisli = Apply<compose, flip,
4                                    Apply<compose, list_bind>>;
```

Code listing 10: Additional monadic functions

Special notice should be brought to the fact that only a single one of those definitions needed to declare a struct at all, all other definitions were possible by functional composition of existing declarations.

4 Conclusion

While at first the concept of pure template meta programming might seem unreasonable, restrictive and overly complicated, this paper shows an orderly and logically structured approach. By defining key features with interoperability in mind, one obtains not only a readable syntax but also reasonably clear representations.

In order to show further that the discussed constructs are not only usable as a theoretical concept, all features explained in this paper and many more, based on the same roots, have been implemented in a library [1]. This can be built by any compiler conforming to the working draft n4659 of the C++ standard, as well as some additional workarounds for using gcc 7.

It is also imaginable that in a very similar manner complex features such as pattern matching or lambda functions can be provided. One could also build a type system on top, for example to support polymorphic monads and a consistent `do` notation.

In its current state, the language has strong functional roots while only missing some key notational improvements that would go far beyond the scope of this paper. Nevertheless, and since such improvements can be built as functional objects themselves, it provides many useful features to meta template programming.

[1] https://github.com/HeroicKatora/CppLibrary

References

[1] Chris Lattner et al. *clang: a C language family frontend for LLVM*. URL: `http://clang.llvm.org/`.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001. ISBN: 978-0-201-70431-0.

[3] Francis Borceux. "Monads". In: *Handbook of Categorical Algebra*. Cambridge: Cambridge University Press, 1994, pp. 186–253.

[4] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. URL: `http://dinosaur.compilertools.net/yacc/`.

[5] Haskell standard library. *Data.List*. URL: `https://hackage.haskell.org/package/base-4.9.1.0/docs/Data-List.html`.

[6] Philip Wadler. "Monads for functional programming". In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24-30, 1995 Tutorial Text*. Ed. by Johan Jeuring and Erik Meijer. Springer Berlin Heidelberg, 1995, pp. 24–52. ISBN: 978-3-540-49270-2.

[7] *Working Draft, Standard for Programming Language C++*. Mar. 21, 2017. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf`.