Lambda Calculus as a Tool for Metaprogramming in C++*

Vít Šefl

Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic
sefl@ksvi.mff.cuni.cz

Abstract. The C++ template system is expressive enough to allow the programmer to write programs which are evaluated at compile time. This can be exploited for example in generic programming. However, these programs are very often hard to write, read and maintain. We introduce a simple translation from lambda calculus to C++ templates and show how it can be used to simplify C++ metaprograms. This variation of lambda calculus is then extended with Hindley-Milner type system and various other features (Haskell-like syntax, user-defined data types, tools for interaction with existing C++ template code) to create a simple functional language. We then build a compiler capable of transforming programs written in this language into C++ template metaprograms.

1 Introduction

Templates were introduced into C++ [6] to facilitate parametric polymorphism. However, the mechanisms behind templates (template specialization, substitution and recently also variadic templates) are powerful enough to express far more complex programs.

Templates form a language within a language. This metalanguage does not have mutable state nor typical control flow operators (if, while, for) and as such is a purely functional language. However, as is usually the case with accidental features, this metalanguage is syntactically cumbersome – both hard to write and hard to read. This poses a maintainability problem for anyone who uses template metaprograms in their C++ code.

We, therefore, turn to traditional functional languages – the first and simplest of them is lambda calculus [4]. Despite its simplicity, it is Turing complete and therefore makes a good foundation for many of today's functional languages. Establishing a connection between lambda calculus and template metalanguage would give us a good basis upon which we could build more expressive and easier to use language.

We design a simple language based on lambda calculus which can be used to describe C++ metaprograms in a clear and simple manner. And finally, we show how such language can be compiled into a template metaprogram.

* This research was supported by the SVV project number 260 453.

The paper is organized as follows. In the next section, we look at other approaches to this problem, including other external tools and specialized metaprogramming libraries. The third section introduces the language, shows how it relates to template metaprograms and gives a short overview of its features. The translation itself is described in the fourth section. The translation of the three basic building blocks of lambda calculus is introduced first, followed by the translation of additional features. The fifth section is a simple summary of the inner workings of the compiler. And finally, the sixth section lists possible ways in which this language could be expanded, ranging from syntactic sugar to type system improvements.

The compiler, language specification and examples are available online [16].

2 Related Work

The idea of using tools to ease up template metaprogramming is not new. Most tools can be divided into two main approaches: C++ libraries and external tools.

C++ libraries that facilitate metaprogramming include examples such as Boost MPL [3] or Boost Metaparse [14]. The aim of these libraries is to provide a simpler way of writing metaprograms while still staying within the limits of C++. However, they still have to deal with the cumbersome syntax of template metaprogramming and thus retain some of the problems related to maintainability.

The other approach – such as ours – is the use of external tools. The main advantage is that these tools do not need to rely on the complexities of template metaprogramming. MetaFun [2] is an example of a tool that transforms simple language into a C++ metaprogram. The translation is even simpler than ours, making heavy use of template-template parameters. However, template-template parameters are at odds with higher order functions. As far as we know, MetaFun does not support currying and lambda abstraction.

Another example is the language EClean [15]. The idea is to translate the input language just enough to be able to interpret it at the level of C++. The compiler therefore produces C++ code that cannot be directly compiled. Instead, it is given to a metaprogram which interprets it. This interpreter is basically a graph rewriting engine, evaluating expressions in a way similar to how compiled Haskell programs are executed. A similar approach is suggested by Porkoláb in [13].

3 Language

Let us take a look at an example of a template metaprogram.

```
template <typename...>
struct pack { };

template <typename, typename>
struct push_front;
```

```
template <typename A, typename... R>
struct push_front<A, pack<R...>>
{ typedef pack<A, R...> type; };

template <template <typename> class, typename>
struct filter;

template <template <typename> class F>
struct filter<F, pack<>>
{ typedef pack<> type; };

template <template <typename> class F,
          typename A, typename... R>
struct filter<F, pack<A, R...>>
{ typedef typename std::conditional<F<A>::value,
    typename push_front<A,
      typename filter<F, pack<R...>>::type>::type,
    typename filter<F, pack<R...>>::type
    >::type type; };
```

The same function, this time in the Haskell [9] language:

```
filter f []    = []
filter f (a:r) =
    if (f a)
    then a:filter f r
    else   filter f r
```

There are some clear parallels. Firstly, defining a function (abstraction) corresponds to declaring a template, arguments are simply the template parameters. Case analysis (determining whether the input list is empty or not) is done by template specialization. Function application is done by template instantiation. And finally, `filter` is recursively applied in both definitions.

Notice that the `filter` template accepts a template-template parameter `F`. This corresponds to an argument with a function type, meaning that template metaprograms are able to express at least a subset of higher-order functions. We shall later see that any higher-order function is expressible, but the translation is more involved.

Rather than taking an existing language and removing features until we can provide reasonable translation to metaprograms, we will start with a simple language. We will then extend it until it covers a reasonable subset of template metaprograms.

One of the simplest functional languages is the untyped lambda calculus. It is a language containing only variables, abstraction and application. This gives us a good basis for two of the basic template metaprogram operations – declaring new templates (abstraction) and instantiation (application).

While basic data types can be represented in lambda calculus (for example by Church encoding), these representations are hard to work with and they are not

very effective. So the first extension is to add numbers and booleans and some operations on them. Thanks to non-type template parameters, we can easily include the underlying `int` and `bool` types together with arithmetic operators (`+`, `-`, `*`, …), logical operators (`&&`, `||`, …) and comparison operators (`==`, `<=`, …).

This can be extended to allow user-defined algebraic data types, with the caveat that these types need custom representation when translated into a template metaprogram. To be able to use values of these data types, we need two operations: construction and case analysis. We opted for case analysis by means of an automatically generated higher-order function (known as an eliminator).

Since lambda calculus only contains expressions, we need to introduce value (and function) bindings. This way, the code can be broken down into smaller, self-contained units. Recursive bindings are allowed.

And finally, the language is extended with a Hindley-Milner type system [5,11]. Full grammar is available in [16]. For example, this gives us enough tools to express the previous `filter` function.

```
data List a = Nil | Cons a (List a);

-- type signature is optional
filter : (a -> Bool) -> List a -> List a;
filter f = list
    (Nil)       -- empty list
    (\x xs ->   -- list with head x, tail xs
        if_ (f x)
            (Cons x (filter f xs))
                    (filter f xs))
```

4 Translation

The `filter` template metaprogram gives a general idea of how the translation is done. However, there is one major issue with the translation of higher-order functions. Consider the identity function, written in the same style as the `filter` function.

```
template <typename T>
struct id { typedef T type; };
```

`id` works perfectly fine when given non-function values:

```
id<int>::type == int
```

But `id` is expected to be able to handle function parameters, such as:

```
id<id>::type == id
```

While some higher-order functions can be expressed, most of them cannot. Templates can express function parameters via the template-template parameter, but `typedef` cannot be used to return such values.

The way to solve this issue is to unify simple types (that can be manipulated easily – both as parameters and as return values via `typedef`) and templates.

Luckily, C++ allows the use of templates inside a class. Each template can therefore be associated with a simple type simply by wrapping the template in another class, which can then be manipulated in the usual way. More flexible identity function can thus be implemented:

```
struct id
{ struct type
  { template <typename T>
    struct apply
    { typedef typename T::type type; }; }; };
```

Some syntactic convenience is lost on the C++ side, but we gain an elegant implementation of higher-order functions. Notice the extra layer of indirection via the `type` structure. While not necessary, it allows us to treat functions and values in a uniform way, which simplifies the translation process.

```
template <int I>
struct Int { static const int value = I; };

struct Two { typedef Int<2> type; };
```

Each metaprogram representing a function has an inner template `apply`, which is the place where the actual parameter can be provided:

```
id::type::apply<Two>::type == Int<2>
```

Notice that the template `apply` expects a structure containing inner type `type`. This is again for consistency reasons – we can treat the application `f x` in the same way no matter what `f` and `x` represent (be it variables bound by abstraction or previously defined constants). A simple helper structure can be defined which does the wrapping on the fly:

```
template <typename T>
struct wrap { typedef T type; };

id::type::apply<wrap<Int<2>>>::type == Int<2>
```

Notice that the previously impossible application is now trivial.

```
id::type::apply<id>::type == id::type
```

This unification of simple types with templates is the basis of the translation. Let us define the translation of untyped lambda calculus first. It suffices to provide the rules for variables, abstraction and application [1].

```
-- variables
translate(x, name) :=
  struct name { typedef typename x::type type; };

-- abstraction
translate(\x -> E, name) :=
  struct name
  { struct type
```

```
    { template <typename x>
      struct apply
      { translate(E, localname)
        typedef typename localname::type type; }; }; };

-- application
translate(M N, name) :=
  struct name
  { translate(M, localname1)
    translate(N, localname2)
    typedef typename
      localname1::type::template apply<localname2>::type
      type; };
```

Interesting thing to note is that this translation also includes value bindings. While giving a name to every subexpression might seem excessive, inner structures have to be defined in most cases anyways. Value bindings can then be translated simply as:

```
translate(x = E) :=
  struct x
  { translate(E, localname)
    typedef typename localname::type type; };
```

Now, let us focus on the extensions. First of all, the language also contains built-in integers and booleans. Translation for literals is fairly simple:

```
translate(True, name) :=
  struct name { typedef Bool<true> type; };

translate(False, name) :=
  struct name { typedef Bool<false> type; };

translate(n, name) :=
  struct name { typedef Int<n> type; };
```

We do not need to explicitly translate operations on these values, they can be implemented in a runtime header and just used from the code as ordinary functions.

Translation of user-defined data types is far more involved. First of all, we need to define the encoding. An algebraic data type contains zero or more constructors, each with zero or more fields. Which constructor was used to create the value can be tracked via a simple integer tag. Fields are then just template parameters. This can even be simplified by translating all constructors to a common variadic template. The type checker will then make sure that we do not try to access non-existent fields.

```
template <int Tag, typename Dummy, typename... Fields>
struct __data { };
```

The extra template parameter `Dummy` is used in template specialization. Its actual value is ignored. For example, the values `Nil` and `Cons` of the previously defined type `List` would be encoded as:

```
__data<0, __dummy>          // tag 0, no fields
__data<1, __dummy, x, xs>   // tag 1, two fields
```

Constructors can be represented as functions taking the appropriate parameters and returning the encoded value.

```
struct Cons
{ struct type
  { template <typename Arg0>
    struct apply
    { struct type
      { template <typename Arg1>
        struct apply
        { typedef __data
            < 1                     // tag
            , __dummy
            , typename Arg0::type   // field 1
            , typename Arg1::type   // field 2
            > type; }; }; }; }; };
```

Case analysis is done via an automatically defined eliminator. For the `List` type, this would be:

```
list : b -> (a -> List a -> b) -> List a -> b
```

One important note is that eliminators for recursive types usually represent the induction scheme for that type. However, since the language can express general recursion, it is more convenient to have eliminators that examine only the top level of a given value.

These eliminators behave in the following way:

```
list z f Nil         = z
list z f (Cons x xs) = f x xs
```

As mentioned previously, template specialization can be used to implement the eliminator.

```
template <typename>
struct apply_alt;

template <typename Dummy>
struct apply_alt<__data<0, Dummy>>
{ // handle empty list
};

template <typename Dummy, typename Arg0, typename Arg1>
struct apply_alt<__data<1, Dummy, Arg0, Arg1>>
{ // handle non-empty list, head is Arg0, tail is Arg1
};
```

```
template <typename Arg>
struct apply
{ // unpack Arg and give it to apply_alt
  typedef typename
    apply_alt<typename Arg::type>::type type; };
```

The rest is just a combination of abstraction and application. The translation of arbitrary data types is simply a natural generalization of this technique. And finally, the let expression – an important part of the Hindley-Milner type system.

```
translate(let x1 = E1; ...; xn = En in E, name) :=
  struct name
  { translate(x1 = E1)
    ...
    translate(xn = En)
    translate(E, localname)
    typedef typename localname::type type; };
```

We have seen that template metaprograms allow recursion. However, the translation cannot be used as-is, because C++ does not allow using types as they are defined. In some cases, this can be solved by a forward declaration, but this cannot be done in general.

Instead, recursion is replaced by a fixed-point combinator behind the scenes. This way, recursion is condensed into a single definition and the rest of the code does not have to deal with the above-mentioned problem.

```
struct fix
{ struct type
  { template <typename f, int i>
    struct apply_rec
    { typedef typename
        f::type::template apply<
            apply_rec<f, i + 1>
          >::type type; };

    template <typename f>
    struct apply
    { typedef typename apply_rec<f, 0>::type
        type; }; }; };
```

The translation of recursive bindings is then simply:

```
f =         \x -> ... f   y ...
f = fix \rec x -> ... rec y ...
```

Interestingly, the fixed-point combinator can also be defined in the language, without the use of recursive bindings.

```
data Rec a = In (Rec a -> a);

out : Rec a -> Rec a -> a;
```

```
out = rec \x -> x;

fix : (a -> a) -> a;
fix = \f -> (\x -> f (out x x)) (In \x -> f (out x x))
```

Instead of defining `fix` every time a recursive binding is encountered, the definition is simply moved into a separate header file. This header file (which can be thought of as the runtime) also contains other common definitions, such as basic arithmetic, logical and comparison operations.

```
template <bool B>
struct Bool { static const bool value = B; };

struct not_
{ struct type
  { template <typename A>
    struct apply
    { typedef Bool<(!A::type::value)> type; }; }; };
```

After the translation, we obtain a C++ header file that can be used with the rest of the code. In the following example, a higher-order function `twice` is translated into C++ code and then used with a template function which adds a level of indirection to a type. More examples can be found in [16].

```
twice : (a -> a) -> a -> a;
twice f = \x -> f (f x)

// Resulting translation.
struct twice { /* ... */ };

struct add_ptr
{ struct type
  { template <typename T>
    struct apply
    { typedef typename T::type* type; }; }; };

int main()
{ std::cout <<
    std::is_same<
      int**,
      twice::type::apply<add_ptr>::
        type::apply<wrap<int> >::type
    >::value; }
```

5 Compilation

The compilation can be broken down into the following steps. Firstly, the source code is ran through a lexer and the result is parsed into an abstract syntax tree. The parsec [8] library is used in both steps. `makeTokenParser` automatically

creates appropriate lexer, which can then be used in the parsing process. Applicative (and monadic) interface allows the definition of parsers in a style very close to the actual BNF grammar.

After we obtain an abstract syntax tree, we remove all recursive bindings and replace them with the `fix` combinator as described above.

The resulting abstract syntax tree is given to the type checker. We are using a variation of Damas-Milner type inference, as described by Typing Haskell in Haskell [7]. The basis of the type inference engine is the type inference monad, a combination of two state monads (one for keeping track of the global substitution, the other for supply for fresh variable names), a reader monad (for various typing contexts and error locations) and finally an error monad.

For user-defined types, the types of each constructor and the eliminator are simply added to the context.

Type signatures are optional. In fact, when checking the validity of type signatures, we first infer the most general type and then attempt to unify it with the declared type. Type checking is successful only if the declared type is a specialization of the inferred type.

Since the semantics of the language does not depend on type information (property known as type erasure), we can just check if type inference was successful and throw the result away.

After that, the abstract syntax tree goes through a renaming process, where each name that does not refer to a top-level entity is replaced with a fresh name. This makes sure that the translated code does not have accidental collisions with C++ keywords and also possible name shadowing problems.

And finally, the abstract syntax tree is transformed into a template metaprogram using the translation given above.

6 Possible extensions

The compiler can be extended in various ways. The language we described in Section 3 was based on Haskell. However, Haskell is a much richer language and its features were proven to be useful by developers and users alike. Let us see which features would be feasible to implement.

In particular, one of the most important features is pattern matching (be it the case expression or function clauses). The translation into template specializations is not straightforward as Haskell makes strong guarantees about the order in which the cases are tried – template specializations do not have such guarantees. This is the reason why we went with explicit eliminators in the end – there is always only one possible template specialization, therefore the ordering problems cannot happen. However, case expressions could be expressed in terms of eliminators.

Haskell also does not require ordering of definitions. It is possible to refer to values defined later, which also allows mutual recursion. Such recursion can seem problematic when all recursion is done only via `fix`. However, there are

fixed-point combinators that can create mutually recursive functions. Consider the following Haskell code:

```
fixmany :: [[a] -> a] -> [a]
fixmany fs = map ($ fixmany fs) fs
[odd, even] = fixmany
    [ \[o, e] n -> if n == 0 then False else e (n - 1)
    , \[o, e] n -> if n == 0 then True  else o (n - 1)
    ]
```

Note that the lambda expressions inside the list get access to both recursive functions that are being defined: the variables `o` and `e` represent the `odd` and `even` functions, respectively. Indeed, in the same way simple recursive definitions are translated in terms of `fix`, mutually recursive functions could be translated in terms of `fixmany`.

Type classes were a consideration but were ultimately dropped. This is one of the reasons why the kind system [9] is limited to kinds of the form `* -> * -> *` (as opposed to, for instance, `(* -> *) -> *`). Without type classes, higher kinded types are not all that useful. The main problem with type classes is that there is no way to translate them into C++ without creating a semantic gap. The use of a type class is implicit in Haskell but would have to be explicit in C++.

Another possible improvement is the module system. Although importing values is possible with the `assume` declaration, this process could be automated. For example, when compiling a module, the compiler could create an interface file containing the types of all defined values which could then be used whenever another file imported that module.

User defined operators would need more work and their usefulness is limited (as overuse of user defined operators usually makes code much less readable). The main problem is that names of operators are usually not valid names in C++. This could be solved by requiring every operator definition to also specify a compiled name. The Haskell fixity declaration could be reused and extended with a naming component:

```
infixl 7 * as times;
infixl 6 + as plus;
```

Another very useful part of Haskell are `where` clauses. Translation of those is very simple (there is a one-to-one correspondence with the let expression) and they make the code more readable.

Haskell also allows for the omission of braces and semicolons via layout. Indeed, indentation is an important part of the syntax, allowing us to distinguish between two separate expressions and one expression that continues over multiple lines simply by observing how are these expressions aligned. Haskell report [9] defines a procedure which converts layout back to explicit braces and semicolons. A similar process could be employed in our case.

Another direction is extending the type system. Since most of the more powerful type systems are built on the same foundation (the semantics of untyped lambda calculus), the type system could easily be extended without changing

the underlying translation. For instance, explicit universal quantification could be added.

GADTs (generalized algebraic data types) [12] do not require new translation and yet add quite a bit of power to the type system. Existential types are a less powerful version, although these are not as useful without type classes.

We could also go further and implement a type system with dependent types, such as Martin-Löf's type theory [10]. Again, the underlying language is the same. After types are erased, the original translation can be used.

The biggest problem we would have to face is to keep type checking decidable; these type systems often need typing hints. When not needed, the values are figured out for instance by unification and then implicitly used. This again creates a semantic gap between the language and the compiled code and having everything be explicit puts an unnecessary burden on the programmer.

7 Conclusion

The compilation process can handle basics of lambda calculus as well as several extensions: let expressions, built-in data types, user-defined data types and recursion. The runtime needed to use metaprograms correctly is also implemented.

The language itself is loosely based on Haskell, the key differences being the lack of layout and several of the more advanced features (type classes, higher kinds). Much like Haskell, the compiler is – thanks to the Hindley-Milner type system – capable of inferring and checking types without explicit annotations.

It is interesting to note that despite being one of the main examples of procedural languages, C++ had a purely functional fragment long before functional features (such as higher-order functions or lambda expressions) became commonplace in mainstream programming languages. This purely functional foundation of templates allowed us to design a simple functional language which can describe a lot of template metaprograms.

The language itself is a compromise – more powerful language features (such as type classes) are more convenient for the programmer but often carry around nontrivial requirements on the compiled code side. We opted for simpler language which allowed us to have straightforward translation without hidden surprises or corner cases. This makes the interaction with existing metaprograms much easier.

Simplicity was also the main goal of implementation – the language can be easily extended, as can be the translation.

This work is also an interesting experiment to see to what extremes can template metaprogramming be taken. In fact, if one is able to use C++ macro machinery to convert string literals into character lists (encoded using either variadic templates or the encoding mentioned earlier), it would not be very hard to use this language to implement its own compiler as a metaprogram.

References

1. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, North-Holland (1984)
2. Érdi, G.: MetaFun: Compile haskell-like code to C++ template metaprograms. `https://gergo.erdi.hu/projects/metafun/` (2011)
3. Gurtovoy, A., Abrahams, D.: Boost.MPL library. `http://www.boost.org/` (2004)
4. Hindley, J.R., Cardone, F.: History of lambda-calculus and combinatory logic. In: Logic from Russell to Church, Handbook of the History of Logic, vol. 5, pp. 723–817. Elsevier (2009)
5. Hindley, R.: The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society 146, 29–60 (1969)
6. ISO: ISO/IEC 14882:2014 Information technology — Programming languages — C++. Tech. rep. (December 2014)
7. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop. Paris (October 1999)
8. Leijen, D., Martini, P.: The parsec library. `https://github.com/haskell/parsec` (2007)
9. Marlow, S., et al.: Haskell 2010 language report (2010)
10. Martin-Löf, P.: Intuitionistic type theory, Studies in Proof Theory, vol. 1. Bibliopolis (1984)
11. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
12. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. SIGPLAN Not. 41(9), 50–61 (Sep 2006)
13. Porkoláb, Z.: Functional programming with C++ template metaprograms. In: Central European Functional Programming School: Third Summer School. pp. 306–353. Springer, Berlin, Heidelberg (2010)
14. Sinkovics, A.: Boost.Metaparse library. `http://www.boost.org/` (2015)
15. Sipos, Á., Zsók, V.: EClean - an embedded functional language. Electronic Notes in Theoretical Computer Science 238(2), 47–58 (2009)
16. Šefl, V.: The norri language: specification, compiler and examples. `https://github.com/vituscze/norri` (2016)