Translating Lambda Calculus into C++ Templates*

Vít Šefl

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

Abstract. The C++ template system is capable of performing arbitrary compile-time computations, which is typically exploited in generic programming libraries. However, the template language itself is syntactically cumbersome. A variety of tools, ranging from libraries to dedicated compilers, was created to alleviate this issue. One such approach is translating a functional program into a template metaprogram. In this work, we present a new way of translating functional programs based on lambda calculus into template metaprograms. The translation produces metaprograms with clearly defined lazy semantics and supports common functional features such as recursion and algebraic data types. We demonstrate its viability by providing a proof-of-concept implementation.

1 Introduction

In C++, templates facilitate parametric polymorphism. The system itself is based on type abstraction, substitution, and specialization, which can be used to express arbitrary computations. Moreover, since templates are evaluated during compilation, they can be used to compute arbitrary values before the program is run. We refer to such computations as *metaprograms* [11]. In addition to compile-time computations, metaprograms are frequently used in generic programming.

The template system forms a language within a language. This sublanguage does not have a mutable state nor any of the typical control flow statements, allowing us to treat it as a simple, purely functional language. However, since metaprogramming is outside of its intended use case, it usually requires a large amount of boilerplate code and other similar syntactic annoyances. Language features such as the `constexpr` keyword seek to provide an alternative but are currently not powerful enough to fully replace template metaprogramming.

Consequently, a variety of tools was created to simplify writing template metaprograms. The approaches vary from libraries that hide some of the boilerplate code [1,2,9] to external tools that allow the programmer to write the code in a different language and then translate it back into a metaprogram [3,6].

Since the language of templates is functional, some tools [3,10] choose a functional language as the source language for the translation. Functional languages

* This work was supported by the Charles University grant SVV-260588

are often based on lambda calculus, whose core concepts match the template mechanisms very closely. However, while higher-order functions can be expressed in terms of *template-template* parameters, these parameters are not flexible and need to be handled separately from the standard type parameters.

In this work, we detail a new direct way of translating lambda calculus into template metaprograms that avoids these pitfalls and is compatible with any standard compliant compiler. We then provide the translation of various features commonly found in functional languages: local bindings, recursion, pattern matching, and algebraic data types, as well as a novel formal treatment of the semantics of the resulting metaprograms. In particular, we show that these metaprograms have well-defined, non-strict semantics.

In order to demonstrate the viability of this translation method, we also provide a proof-of-concept compiler for a simple functional language based on this work. The language uses Hindley-Milner type system and its syntax is inspired by Haskell. [1]

This work is organized as follows. In the next section, we discuss other approaches to this problem. The third section gives a brief overview of template metaprogramming in C++. The core translation is laid out in the fourth section and the translation of the additional features in the fifth section. The sixth section details the semantics of the resulting metaprograms. The final section provides examples of integration with regular C++ code.

2 Related Work

The most prominent examples of C++ libraries that facilitate metaprogramming are Boost Hana [2] and Boost Metaparse [9]. These libraries aim to provide an easier and more convenient way of writing template metaprograms. Boost Hana provides a general framework for writing metaprograms, while Boost Metaparse functions specifically as a parser generator. However, since these libraries still operate within C++ itself, they cannot be used to eliminate all boilerplate code.

The other approach is the use of external tools. The main advantage is that these tools hide most of the complexities of template metaprogramming from the programmer.

MetaFun [3] is an example of a tool that translates a simple functional language into metaprograms. The translation is straightforward, making use of template-template parameters to express higher-order functions. To our knowledge, this tool is not capable of expressing currying or lambda abstraction.

This approach to metaprogramming is the closest to the approach chosen by this work. A major advantage is that the resulting metaprograms remain legible to C++ programmers and can, if necessary, be adjusted manually. This flexibility is invaluable when the generated metaprograms need to interact with existing metaprograms, which are generally not immediately compatible.

EClean [10] uses a more complicated process of translation. The input language is translated into an intermediate language, which is then interpreted by

[1] `https://github.com/vituscze/norri`

a template metaprogram. This interpreter is a graph rewriting engine that evaluates expressions similarly to how compiled Haskell code is executed. A similar approach is suggested by Porkoláb [5].

Note that the translation of the source language can also be performed by a metaprogram. As an example, this hybrid approach is used to extend C++ with a self-contained domain specific language [7].

These methods typically sacrifice clarity and transparency of the translated metaprograms in order to improve their efficiency or to reduce the dependency on third party tools. As mentioned previously, this opaqueness might be undesirable in some situations.

3 Template Metaprogramming

C++ templates facilitate parametric polymorphism. However, when combined with other language constructs such as `static const` or `using`, templates become expressive enough to describe arbitrary compile-time computations. A brief overview of this concept is given in this section.

Listing 1 shows a standard use case of templates. The class definition is parameterized over the element type. This template can then be instantiated with a concrete type and used as a regular data type.

```
template <typename T>
class vector {
  T& operator[](size_t index);
};

using int_vector = vector<int>;
```

Listing 1. Parametric polymorphism in C++

The strength of the template system lies in the ability to define compile-time constants that depend on the template parameter. A template can thus be treated as a function, where the input is the template parameter and the output is the defined constant. The result of a metaprogram is obtained by instantiating the template with the desired arguments.

Compile-time constants can be defined with a `using` statement (for type constants) or as `static const` class members (for value constants). There are other ways of defining compile-time constants (such as enumeration labels), but they are interchangeable as far as template metaprogramming is concerned.

Some values can be promoted to the type level, which allows the metaprogram to treat its inputs uniformly as type parameters. Listing 2 shows how to promote `int` constants. The value is accessed by referring to the `static const` member.

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

using int_array = std::array<int, Int<5>::value>;
```

Listing 2. Type-level promotion

A template may also be specialized, providing a more specific definition for a subset of template parameters, which can be used by metaprograms to perform case analysis. The template definition is chosen based on how well the arguments fit the specialization, rather than trying the definitions in some predetermined order. Listing 3 combines template specialization and recursion to implement type-level factorial.

```
template <typename T>
struct factorial;

template <>
struct factorial<Int<0>> {
  using type = Int<1>;
};

template <int N>
struct factorial<Int<N>> {
  using type = Int<N * factorial<Int<N - 1>>::type::value>;
};

factorial<Int<3>>::type::value == 6
```

Listing 3. Factorial function

Similarly, variadic templates may be used to represent ordered sequences and template-template parameters to represent higher-order functions. Listing 4 shows an example of such a function. The first parameter represents a template function, which is then applied twice to the second parameter. To treat a qualified name that depends on a template parameter as a type (template), the `typename` (`template`) keyword must be used.

```
template <template <typename> class F, typename X>
struct twice {
  using type = typename F<typename F<X>::type>::type;
};

twice<factorial, Int<3>>::type::value == 720
```

Listing 4. Higher-order function

4 Translating Lambda Calculus

Lambda calculus is a simple functional language. Template metaprogramming and lambda calculus share some core concepts but the correspondence is not perfect. In this section, we show possible ways of representing lambda calculus as template metaprograms and discuss their advantages and disadvantages. We select one representation to be used as the basis of the translation.

A lambda calculus expression can be either a variable, an abstraction, or an application. The basic idea is to use template parameters or type names as

variables, template definition as an abstraction, and template instantiation as an application.

Notice that the code in Listing 4 needs to know which parameters represent a function. However, lambda calculus generally makes no distinction between functional and non-functional parameters.

One option is to consider only the simply-typed lambda calculus. In this variation of lambda calculus, each variable has a concrete type and can thus be used to distinguish between functional and non-functional parameters. The functional parameters can then be expressed as template-template parameters of the appropriate nesting and every other parameter as a regular type parameter.

The main downside of this approach is that a single expression needs to be translated into multiple template metaprograms, one for each combination of parameter arities. Another issue is that template-template parameters cannot be used directly with the `using` statement. Instead, the template structure needs to be reconstructed whenever such a parameter is encountered.

The other option is to unify regular and templated types. Every template parameter can then be treated uniformly as a type. Each template can be associated with a simple type by wrapping the template in another class. The identity function defined in Listing 5 shows an example of this unification. Note that this self-application would not be possible with template-template parameters.

Another advantage of this approach is that the resulting translation is type-agnostic, and may be used with both typed and untyped source languages.

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

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

Listing 5. Flexible identity function

The inner class `type` provides a layer of indirection, which is necessary to handle self-referential expressions as well as to simplify the translation of additional features. A direct translation of self-referential expressions would lead to an invalid C++ code due to the use of incomplete types.

The downside is that simple types cannot be used as metaprogram arguments. Instead, these types need to be wrapped in another class. Listing 6 shows a wrapping class and its use with the previously defined identity function.

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

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

Listing 6. Argument wrapping

The full translation of lambda expressions is given in Listing 7. The translation of variables and abstractions matches the earlier translation of the identity function. The translation of applications requires the use of inner classes. The names of these classes, S_1 and S_2, must be unique to prevent name collisions. Similarly, since templates do not allow parameter name shadowing, variables must be fresh.

Note that the class S_1 is not strictly necessary and could be removed by changing the name of the inner definition in the translation of the expression E_1. For simplicity, we do not present this optimization here.

```
translate(x) ≝
  using type = typename x::type;

translate(λx.E) ≝
  struct type {
    template <typename x>
    struct apply { translate(E) };
  };

translate(E1 E2) ≝
  struct S1 { translate(E1) };
  struct S2 { translate(E2) };
  using type = typename S1::type::template apply<S2>::type;
```

Listing 7. Lambda expression translation

5 Translating Functional Languages

Pure lambda calculus lacks many features of modern functional languages that make programming more convenient and better tractable. In particular, the program cannot be structured into multiple named expressions and data needs to be encoded as functions. In this section, we address this issue by providing a translation of bindings, recursive definitions, and data types.

5.1 Bindings

A binding is used to associate an expression with a name, which can be used to break the program apart into small reusable definitions. Since metaprograms are already associated with a type name, the translation simply wraps the definition inside an appropriately named class. Local bindings, which are used to name subexpressions, use identical translation. The translation is shown in Listing 8.

```
translate(x = E) ≝
  struct x { translate(E) };
```

Listing 8. Binding translation

5.2 Recursion

While recursion can be accomplished with the use of a fixed-point combinator, recursive bindings are more convenient to work with.

A template may recursively refer to itself, which can be used to directly translate recursive bindings of the form $x = \lambda y.E(x)$. However, recursive bindings of the form $x = x$ or $x = E_1(x)\,E_2(x)$ present a problem. The translation of the recursive occurrences of x requires the definition of x`::type` which is not available at that point.

One option is to restrict the recursion to functions only. The expression in the problematic bindings may then be η-expanded to $\lambda y.x\,y$ or $\lambda y.E_1(x)\,E_2(x)\,y$.

The other option is to translate the recursive bindings in two steps. In the first step, the recursive bindings are replaced with regular bindings by adding fixed-point combinators. Regular bindings are then translated using the techniques described earlier. This process is described in Listing 9.

$$\text{translate}(x = E(x)) \stackrel{\text{def}}{=} \text{translate}(x = Y\,\lambda r.E(r))$$

Listing 9. Recursive binding translation

The choice of the fixed-point combinator is not important. We have used the Y combinator which is defined as $\lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$. It is sufficient to translate the combinator just once and then refer to it from the rest of the code.

The same result can be accomplished with a handwritten, directly recursive combinator, such as the one shown in Listing 10. This particular implementation is optimized to produce as few nested types and template instantiations as possible.

The main advantage of this approach is its flexibility. There is a large variety of fixed-point combinators that can be used to translate more complex recursion schemes, such as mutual recursion. The direct translation cannot be used in this case because C++ does not allow forward declarations of nested classes.

```
struct fix {
  struct type {
    template <typename F>
    struct apply {
      using type = typename
        F::type::template apply<apply<F>>::type;
    };
  };
};
```

Listing 10. Fixed-point combinator

5.3 Simple Data Types

C++ templates can use non-type parameters in their definition. One subset of these non-type parameters are the values of integral and enumeration types. Such values can be promoted to the type level and then used as regular type

parameters. For example, if `Int` is the type promoted version of `int`, then an integer constant can be translated as shown in Listing 11. The values of other data types can be translated similarly.

translate(n) $\stackrel{\text{def}}{=}$

```
  using type = Int<n>;
```

Listing 11. Integer translation

However, standard operators cannot be applied to these type-promoted constants. Instead of translating these operators directly, it might be preferable to collect their implementation into a separate header file to reduce the amount of generated code. The header can then be included with the rest of the translated code. As an example, Listing 12 shows an implementation of boolean negation.

```
struct not_ {
  struct type {
    template <typename B>
    struct apply {
      using type = Bool<!B::type::value>;
    };
  };
};
```

Listing 12. Boolean negation

5.4 Complex Data Types

Simple data types use unary templates with a non-type parameter to store one value of integral type. This approach can be extended to more complex data types by using templates with more parameters. For example, any template with two type parameters can be used to represent type-level pairs. However, as with simple data types, the non-trivial task is implementing operations to manipulate the values of such data types.

Instead of focusing on a particular data type, we describe the translation of a class of data types known as algebraic data types. An algebraic data type is a data type formed as a combination of products (tuples) and sums (variants). These data types, therefore, include all records (tuples without any variants) and enumerations (variants without any tuples).

For each data type, we need to specify how its values are represented, constructed (introduced), and deconstructed (eliminated). Let the data type D consist of m variants. Let $D_i(f_1, \ldots, f_{n_i})$ be a value of D, where D_i is the variant and f_1 to f_{n_i} are the fields.

Representation The values can be represented in two ways. Each variant D_i can be represented as a unique template with n_i type parameters. If the variant has no fields, a non-templated type is used instead.

The other approach is to use one variadic template with one non-type parameter and a variable number of type parameters. The non-type parameter determines the variant and the other type parameters are the fields. Listing 13 shows such a template.

```
template <int Variant, typename... Fields>
struct data { };
```

Listing 13. Algebraic data type representation

These two representations behave identically in normal situations, but differ slightly when misused.

Construction A value of D is constructed by picking the desired variant D_i and providing a value for each field. The translation is shown in Listing 14. Like before, the names S_j need to be unique.

translate$(D_i(E_1, \ldots, E_{n_i})) \stackrel{\text{def}}{=}$
  $\forall j \in \{1, \ldots, n_i\}$
    `struct` S_j `{` translate(E_j) `};`
  `using type = data<`i`, ..., typename` S_j`::type, ...>;`

Listing 14. Constructor translation

Instead of constructing D_i directly, it might be preferable to use $\lambda x_1 \ldots x_{n_i}.\ D_i(x_1, \ldots, x_{n_i})$, which can be partially applied and used with higher-order functions.

Deconstruction The values of D are deconstructed by performing a case analysis. The input of the case analysis is an expression E which represents some value of the data type D. Each case is described by a clause which is a pair consisting of a pattern pat_i and an expression E_i. A pattern can either be a *wildcard pattern* (represented by an underscore) or a *variant pattern* D_j followed by a sequence of distinct variables x_1 to x_{n_j}. The expression E_i may refer to the variables that appear in pat_i.

We require the patterns to be distinct (up to variable renaming) and the case analysis to be complete (if a variant does not have a corresponding variant pattern, a wildcard pattern must be present).

Case analysis proceeds by evaluating E to a value $D_i(f_1, \ldots, f_{n_i})$ for some i. Next, the corresponding clause $pat_j \to E_j$ is selected and, if applicable, the variables x_1 to x_{n_i} are bound to the values of fields f_1 to f_{n_i}. The result of the case analysis is then the value of the expression E_j.

A wildcard pattern is selected only when no matching variant pattern is found, which guarantees that the selection of a clause is unique thanks to the distinctness and completeness conditions above.

Case analysis can be translated as a template with one type parameter. The definition of this template consists of a template specialization for each of the clauses. The full translation is shown in Listing 15.

translate(case E {$pat_1 \to E_1, \ldots, pat_p \to E_p$}) $\stackrel{\text{def}}{=}$

```
template <typename>
struct _case;
```

$\forall i \in \{1, \ldots, p\}$
  translate($pat_i \to E_i$)

```
struct S { translate(E) };
using type = typename _case<typename S::type>::type;
```

translate($D_i\, x_1 \ldots x_{n_i} \to E$) $\stackrel{\text{def}}{=}$

```
template <typename f1, ..., typename fni>
struct _case<data<i, f1, ..., fni>> {
```
  $\forall j \in \{1, \ldots, n_i\}$
    `struct` x_j `{ using type =` f_j`; };`
  translate(E)

```
};
```

translate(_ $\to E$) $\stackrel{\text{def}}{=}$

```
template <typename>
struct _case {
```
  translate(E)

```
};
```

Listing 15. Deconstructor translation

If a variant contains no fields, the corresponding template specialization is a full specialization. An example of full template specialization is shown in Listing 16.

```
template <>
struct _case<data<0>> { };
```

Listing 16. Full template specialization

Until C++17, a full specialization of a class could only occur at the namespace level. When working with older C++ compilers, only partial specialization should be used, which can be accomplished by adding an extra type parameter to the `data` template. The value of this parameter is irrelevant since it is never used.

As presented, the `_case` template cannot distinguish between two variants of different data types. This is not a problem if the source language can guarantee that case analysis is only performed on the correct values. If no such guarantee exists, it is preferable to represent each variant with a unique template instead of using the generic `data` template.

A more complex case analysis with overlapping cases or nested patterns can be implemented in terms of the simple case analysis given here [4].

As an example, Listing 25 uses this encoding on a singly-linked list.

6 Semantics

In order to show that the translated metaprograms behave in a consistent way, we first only consider strongly normalizing expressions of the source language (expressions whose reduction always terminates). We then show that these metaprograms reduce in normal order. Note that this section only accounts for the relevant portion of the underlying template model [8].

6.1 Preservation

Consider a well-behaved expression in the source language. We need to show that the translation preserves reduction. In particular, we need to consider function application, local bindings, and case analysis. The reduction behavior of operations on promoted data types, once fully applied, is simply given by the underlying C++ computational model. A step-by-step explanation is also available. [2]

Function Application Reduction of function application is given by the β-rule $(\lambda x.M)N \rightsquigarrow M[x := N]$. The translated metaprogram (Listing 17) unpacks the inner `type` of the lambda abstraction and then instantiates the inner template `apply`, which contains the translation of M. The instantiation replaces all free occurrences of x with the class S_2. Notice that these variables now refer to S_2`::type` which is the translation of N. Thus, the resulting metaprogram matches the translation of $M[x := N]$.

```
translate((λx.M) N) =
  struct S1 {
    struct type {
      template <typename x>
      struct apply { translate(M) };
    };
  };
  struct S2 { translate(N) };
  using type = typename S1::type::template apply<S2>::type;
```

Listing 17. Reduction of function application

Note that since we require variables to be fresh, the substitution does not have to consider the capture of free variables or variable shadowing.

Local Bindings The reduction of non-recursive local bindings is given by $\mathrm{let}\, x = N \,\mathrm{in}\, M \rightsquigarrow M[x := N]$. We can see that the free occurrences of x in the translation of M (Listing 18) directly refer to the translation of N and the resulting metaprogram thus matches the translation of $M[x := N]$.

[2] `https://github.com/vituscze/norri/blob/master/semantics.md`

```
translate(let x = N in M) =
  struct x { translate(N) };
  translate(M)
```

Listing 18. Reduction of local bindings

Case Analysis The reduction of the case analysis (deconstruction) is given by case $(D_i(N_1, \ldots, N_j))\,\{\ldots, D_i\,x_1\,\ldots\,x_j \to M, \ldots\} \rightsquigarrow M[x_1 := N_1, \ldots, x_j := N_j]$. If the constructor tag does not match any of the patterns, the wildcard pattern, which must be present, is used. Without loss of generality, we only consider the case of a unary constructor.

The translated metaprogram (Listing 19) constructs the encoded value of the algebraic data type in the class S. `S::type` contains the constructor tag i and its second parameter refers to the translation of N.

```
translate(case (D_i(N)) {..., D_i x → M, ...}) =
  template <typename>
  struct _case;

  template <typename f>
  struct _case<data<i, f>> {
    struct x { using type = f; };
    translate(M)
  };

  struct S {
    struct S_1 { translate(N) };
    using type = data<i, typename S_1::type>;
  };

  using type = typename _case<typename S::type>::type;
```

Listing 19. Reduction of case analysis

`S::type` is then given to the template class `_case`. Since the constructor tags are unique across the template specializations, the encoded value matches at least one specialization (case analysis is guaranteed to cover all cases) and at most two specializations (one with a matching tag and one wildcard).

In case there is only a single match, C++ has no choice but to use that match. When there are two matches, C++ prefers the more specific match, which is the specialization with the matching constructor tag. In either case, the correct template specialization is selected.

Once the correct specialization is instantiated, the value stored in the encoded constructor is wrapped in the class x and the final result is the translation of M, which can refer to the translation of N via the variable x. This result matches the translation of $M[x := N]$.

6.2 Evaluation Order

Two Phase Compilation Template code is compiled in two phases. In the first phase, the compiler only processes the parts of the code that do not depend on the template parameters. No instantiation takes place at this time. This phase ensures the template is well-formed, even if it is never used.

The second phase occurs when the template is used with concrete arguments. This forces the instantiation of the template, substituting the template parameters with the given arguments. Code that depends on those parameters can be processed at this time.

In some cases, first phase processing might be undesired. As an example, `static_assert` which unconditionally fails with a given message can be used to give clearer error messages to partial functions. However, such assertion would be triggered during the first phase processing, before the function is even used.

First phase processing can be avoided by tricking the compiler into assuming the code depends on the parameter. The template `always_false` in Listing 20 does not depend on the template parameter, but to see that, the compiler needs to instantiate the template. As a result, the static assert in the template `succeeds` does not see that its parameter is false during the first phase processing and the assert is only triggered when the outer template is instantiated during the second phase processing.

```
template <typename T>
struct always_false
{ static const bool value = false; };

template <typename T>
struct fails {
  static_assert(false);
};

template <typename T>
struct succeeds {
  static_assert(always_false<T>::value);
};
```

Listing 20. Fake parameter dependency

Instantiation C++ templates distinguish between implicit and explicit instantiation. Implicit instantiation occurs when a code refers to the template in a context that requires its definition. Explicit instantiation occurs as a result of a special instantiation statement.

```
template <typename T>
struct s { using type = int; };

s<int>::type x = 5;       // implicit
template struct s<int>;   // explicit
```

Listing 21. Implicit and explicit instantiation

Explicit instantiation of a class template forces the instantiation of all its members, whereas implicit instantiation only instantiates whatever is necessary. In other words, implicit instantiation is lazy.

Laziness The translation exploits the previous observation by using the inner type name `type`. The classes are set up in such a way where referring to the class itself does not force instantiation of any of its members. Referring to the inner type name `type` forces their instantiation, which drives the evaluation. Another benefit of this approach is that the translation does not need fake parameter dependencies.

This difference is best exemplified on the encoding of algebraic data types. Notice that the arguments passed to the `data` template are of the form x`::type` for some x. This observation suggests that those data types are strict. And indeed, a C++ compiler will quickly hit the template instantiation limit when trying to compile an infinite data structure.

However, we are not forced to access the inner `type` member when creating an encoded value. We can change the translation of the constructor from Listing 14 and the the pattern from Listing 15 as follows.

```
translate(D_i(E_1, ..., E_{n_i})) ≝
  ∀j ∈ {1, ..., n_i}
    struct S_j { translate(E_j) };
  using type = data<i, ..., S_j, ...>;

translate(D_i x_1 ... x_{n_i} → E) ≝
  template <typename x_1, ..., typename x_{n_i}>
  struct _case<data<i, x_1, ..., x_{n_i}>> {
    translate(E)
  };
```

Listing 22. Non-strict algebraic data types

And indeed, when a metaprogram is translated using this modification, it can create and operate on infinite data structures. Value recursion also functions as expected.

While the translated metaprograms are lazy, it is also possible to force strict evaluation. For example, the `seq` operation from Haskell's `Prelude`, which forces the evaluation of its first argument and then returns the second one, can be implemented by translating $\lambda xy.y$ and replacing the translation of y by the code in Listing 23.

```
template <typename X, typename Y>
using _snd = Y;

using type = _snd<typename x::type, typename y::type>;
```

Listing 23. The `seq` operation

Notice that when translating top-level bindings, the translated metaprograms are not contained in any template. As a result, they will be evaluated during

compilation regardless of whether they are used. If necessary, we can simply wrap these metaprograms inside a template class as shown in Listing 24.

```
template <typename _T>
struct tmp_impl {
  translate(x_1 = E_1; ...; x_n = E_n)
};

using tmp = tmp_impl<void>;
```

Listing 24. Top-level template wrapping

6.3 Compilation Errors

Translated metaprograms produce error messages during compilation if their reduction gets stuck or does not terminate. As a result, well-behaved expressions in the source language translate into metaprograms that do not produce error messages. Thanks to lazy evaluation, this guarantee extends even to expressions that are well-behaved only under a certain evaluation order.

However, C++ compilers impose a limit on the template instantiation depth, which can result in compilation failure even for well-behaved metaprograms. Compilers typically emit a specific error which makes this issue easy to diagnose. If necessary, compiler flags can be used to increase this limit (for example `-ftemplate-depth` in GCC).

While the translation itself does not avoid compilation errors, most of these errors can be removed by restricting which expressions are valid in the source language. For example, simply-typed lambda calculus is strongly normalizing and its reduction does not get stuck. All expressions are thus well-behaved and if translated, the resulting metaprogram can only fail to compile due to the template instantiation depth limit.

In essence, template compilation errors can be transformed into type errors in the source language. Such errors are much easier to understand and correct.

7 Practical Examples

In this section, we provide two examples of combining the resulting metaprograms with existing metaprogramming code.

The translated metaprograms can often be used directly. Nevertheless, an auxiliary metaprogram can simplify the code, such as when manipulating encoded data types. Instead of a list, we might wish to use a pack of template parameters. This representation is not only more succinct, but it also allows the pack to be expanded into expressions.

Suppose that a strict list data type consists of a nullary variant *Nil* and a binary variant *Cons*, and the encoding uses explicit names instead of the generic `data` template. Listing 25 shows a conversion between such lists and template parameter packs. Note that template parameter packs are not first-class citizens of C++ and must, therefore, be wrapped in an auxiliary template

`pack`. The `add` metaprogram adds a new element to a template parameter pack. The list encoding is recursively constructed by `to_list` and deconstructed by `from_list`.

```
template <typename...>        struct pack;
template <typename, typename> struct add;
template <typename...>        struct to_list;
template <typename>           struct from_list;

template <typename T, typename... U>
struct add<T, pack<U...>> {
  using type = pack<T, U...>;
};

template <>
struct to_list<> {
  using type = Nil;
};

template <typename T, typename... U>
struct to_list<T, U...> {
  using type = Cons<T, typename to_list<U...>::type>;
};

template <>
struct from_list<Nil> {
  using type = pack<>;
};

template <typename T, typename U>
struct from_list<Cons<T, U>> {
  using type =
    typename add<T, typename from_list<U>::type>::type;
};
```

Listing 25. List conversion

Similarly, existing metaprograms can be adapted for use in higher-order functions. Unary predicates from the `type_traits` header can be adapted as shown in Listing 26. This process can be automated and extended for predicates and functions of higher arity.

```
template <template <typename> class F>
struct predicate {
  template <typename T>
  struct apply {
    using type = Bool<F<typename T::type>::value>;
  };
};
```

Listing 26. Type function conversion

7.1 Precomputation

Since metaprograms are evaluated during compilation, they can be used to precompute constants. The main advantage of this approach is that the computation can be parametrized, which is especially useful when multiple constants depend on a small set of input parameters.

One example is the precomputation of small prime numbers, which is useful when generating large prime numbers. This computation comes with a natural tradeoff: the more time we spend precomputing primes during compilation, the less time we spend finding primes during run time. The metaprogram generates prime numbers smaller than a given value, which gives us control over the tradeoff.

$$
\begin{aligned}
&\mathit{diff} = \lambda \mathit{step}\ \mathit{start}\ \mathit{list}.\ \text{case}\ \mathit{list}\\
&\quad \{\mathit{Nil} \to \mathit{Nil}\\
&\quad , \mathit{Cons}\ x\ \mathit{xs} \to \text{case}\ \mathit{compare}\ x\ \mathit{start}\\
&\qquad \{\mathit{LT} \to \mathit{Cons}\ x\ (\mathit{diff}\ \mathit{step}\ \mathit{start}\ \mathit{xs})\\
&\qquad , \mathit{EQ} \to \mathit{diff}\ \mathit{step}\ (\mathit{start}\ +\ \mathit{step})\ \mathit{xs}\\
&\qquad , \mathit{GT} \to \mathit{diff}\ \mathit{step}\ (\mathit{start}\ +\ \mathit{step})\ (\mathit{Cons}\ x\ \mathit{xs})\\
&\qquad \}\\
&\quad \}\\
\\
&\mathit{sieve} = \lambda \mathit{list}.\ \text{case}\ \mathit{list}\\
&\quad \{\mathit{Nil} \to \mathit{Nil}\\
&\quad , \mathit{Cons}\ x\ \mathit{xs} \to \mathit{Cons}\ x\ (\mathit{sieve}\ (\mathit{diff}\ x\ x^2\ \mathit{xs}))\\
&\quad \}\\
\\
&\mathit{between} = \lambda x\ y.\ \text{case}\ x\ \leq\ y\\
&\quad \{\mathit{False} \to \mathit{Nil}\\
&\quad , \mathit{True} \to \mathit{Cons}\ x\ (\mathit{between}\ (x\ +\ 1)\ y)\\
&\quad \}\\
\\
&\mathit{primes} = \lambda n.\ \mathit{sieve}\ (\mathit{between}\ 2\ n)
\end{aligned}
$$

Listing 27. Sieve of Eratosthenes

The metaprogram is based on the sieve of Eratosthenes. The generated primes are stored in a list, which is then used to initialize an array. The definitions can be found in Listing 27. The $\mathit{compare}\ x\ y$ expression answers whether x is less than, equal to, or greater than y.

The list of prime numbers is obtained by applying the *primes* function to the upper bound. The function generates a list of candidate numbers up to the bound and then performs the sieve operation. At each step, the head of the list x is marked as a prime number, and multiples of x (starting with x^2) are removed from the remainder of the list.

Once these definitions are translated into a C++ metaprogram, the resulting list can be accessed as `primes::type::apply<wrap<Int<n>>>::type` (for some number `n`).

The list cannot be used to directly initialize an array. Instead, the list needs to be converted into a template parameter pack, which can then be expanded

into the array initializer. A template parameter pack `T`, whose elements are types that contain a `value` constant, can be expanded via {`T::value...`}. Listing 28 details how to automate this process.

```
template <typename>
struct to_array;

template <typename... T>
struct to_array<pack<T...>> {
  static const int size = sizeof...(T);
  static const int data[];
};

template <typename... T>
const int to_array<pack<T...>>::data[] = {T::value...};
```

Listing 28. Array initialization using a template parameter pack

And finally, Listing 29 shows how to combine these operations to initialize and use a precomputed array of prime numbers.

```
using list  = primes::type::apply<wrap<Int<50>>>::type;
using array = to_array<to_pack<list>::type>;
for (int i = 0; i < array::size; ++i)
  std::cout << array::data[i] << "␣";
```

Listing 29. Precomputed array usage

7.2 Generic Programming

Template metaprograms are commonly employed in generic programming. As an example, metaprograms from the `type_traits` header are frequently used to check the prerequisites of the elements of standard library containers. These operations range from checking whether the element type supports a given operation to manipulating types in an iterator definition.

Suppose we want to create a pool allocator for a set of types. A pool allocator allocates a large chunk of memory at the start, which is then divided evenly into blocks large enough to hold a value of any of the given types. Allocation proceeds by finding an empty block and returning it. Empty blocks can be tracked using a linked list, which allows the operations to function in $\mathcal{O}(1)$ time.

The allocator first ensures that all types meet the given criteria. The *check* function is used for this task. Its input is a list of predicates and a list of types. The check succeeds if all types satisfy all predicates. It is implemented in terms of the *all* function, which checks whether all elements of a list satisfy one predicate.

The block size is also calculated during compilation. The *blockSize* function computes the size of the block as the maximum size among the input types, which is then rounded to the nearest power of two by the *nextPower* function. The expression $max\, x\, y$ denotes the maximum of x and y. The *size* function computes the size of its input.

The definitions can be found in Listing 30. Two auxiliary functions are used: *foldr* and *loop*. The *foldr* function combines all elements of a list into a single value, using a combining function and an initial value. The *loop* function repeatedly applies a given function to a value while a condition holds.

$$
\begin{aligned}
&foldr = \lambda f\, z\, list.\ \text{case}\, list \\
&\quad \{Nil \to z \\
&\quad , Cons\, x\, xs \to f\, x\, (foldr\, f\, z\, xs) \\
&\quad \}
\end{aligned}
$$

$$
\begin{aligned}
&loop = \lambda p\, f\, x.\ \text{case}\, p\, x \\
&\quad \{False \to x \\
&\quad , True \to loop\, p\, f\, (f\, x) \\
&\quad \}
\end{aligned}
$$

$$
\begin{aligned}
&all = \lambda p.\, foldr\, (\lambda x\, r.\, p\, x\ \wedge\ r)\, True \\
&check = \lambda ps\, xs.\, all\, (\lambda p.\, all\, p\, xs)\, ps \\
&nextPower = \lambda n.\, loop\, (\lambda x.\, x\ <\ n)\, (\lambda x.\, 2x)\, 1 \\
&blockSize = \lambda ts.\, nextPower\, (foldr\, (\lambda x\, r.\, max\, (size\, x)\, r)\, 1\, ts)
\end{aligned}
$$

Listing 30. Type check and block size calculation

The *size* function is defined separately as an auxiliary metaprogram in order to use the `sizeof` operator. The definition can be found in Listing 31.

```
struct size {
  struct type {
    template <typename T>
    struct apply {
      using type = Int<sizeof(typename T::type)>;
    };
  };
};
```

Listing 31. Type size metaprogram

Listing 32 shows a possible definition of such an allocator. For the sake of brevity, the allocator only checks one predicate (`std::is_pod`). The predicate list `predicates` and the type list `types` are passed to the translated `check` metaprogram and its result is used in `static_assert`, which halts the compilation and reports the specified error message if the check fails.

Similarly, the type list `types` is passed to the translated `blockSize` metaprogram and its result is used to initialize the `block_size` constant, which can then be used in the appropriate allocation operation.

```
template <typename... T>
struct allocator {
  using predicates = to_list<predicate<std::is_pod>>;
  using types      = to_list<T...>;
  static_assert(check::type::apply<predicates>
                    ::type::apply<types>
                    ::type::value);
```

```
  static const int block_size =
    blockSize::type::apply<types>::type::value;
};
```

Listing 32. Allocator for a set of types

8 Conclusion

While recent C++ standards offer more options for performing compile-time computations thanks to the `constexpr` keyword, the support of type-level programming is still lacking. Template metaprogramming thus remains an important tool for implementing generic data structures and functions. However, template metaprograms are often hard to read and write. Some tools seek to alleviate these problems by translating functional code into metaprograms.

We present a new way of translating functional code based on lambda calculus into template metaprograms. The main advantages of our method are its simplicity and well-defined, non-strict semantics. The translation uses direct rules and the resulting metaprograms can be easily incorporated into existing C++ code. The translation can be used with both typed and untyped languages, and also includes bindings, recursion, and complex data structures.

Since the source language is based on an existing programming paradigm, a wealth of existing programming techniques can be reused. Similarly, the source language can be subject to existing optimizing transformations. We hope that this work encourages programmers to write more complex metaprograms as well as to simplify the existing ones.

References

1. Abrahams, D., Gurtovoy, A.: Boost.MPL library. `http://www.boost.org/` (2004)
2. Dionne, L.: Boost.Hana library. `http://www.boost.org/` (2020)
3. Érdi, G.: MetaFun: Compile Haskell-like code to C++ template metaprograms. `https://gergo.erdi.hu/projects/metafun/` (2011)
4. Maranget, L.: Two Techniques for Compiling Lazy Pattern Matching. Tech. rep., INRIA (1994)
5. Porkoláb, Z.: Functional Programming with C++ Template Metaprograms. In: Central European Functional Programming School: Third Summer School. pp. 306–353. Springer (2010)
6. Porkoláb, Z., Sinkovics, Á.: Expressing C++ Template Metaprograms as Lambda Expressions, pp. 97–111. Intellect (2009)
7. Porkoláb, Z., Sinkovics, Á., Siroki, I.: DSL in C++ Template Metaprogram, vol. 8606, pp. 76–114. Springer (2015). https://doi.org/10.1007/978-3-319-15940-9
8. Siek, J., Taha, W.: A semantic analysis of C++ templates. In: Thomas, D. (ed.) ECOOP 2006 – Object-Oriented Programming. pp. 304–327. Springer (2006)
9. Sinkovics, A.: Boost.Metaparse library. `http://www.boost.org/` (2020)
10. Sipos, Á., Zsók, V.: EClean - An Embedded Functional Language. Electronic Notes in Theoretical Computer Science **238**(2), 47–58 (2009)
11. Veldhuizen, T.: Expression templates. C++ Report **7**, 26–31 (1995)