Implementing Monads for C++ Template Metaprograms

Ábel Sinkovics, Zoltán Porkoláb

Dept. of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
E-mail: {`abel`|`gsd`}`@elte.hu`

Abstract

C++ template metaprogramming is used in various application areas, such as expression templates, static interface checking, active libraries, etc. Its recognized similarities to pure functional programming languages – like Haskell – make the adoption of advanced functional techniques possible. Such a technique is using monads, programming structures representing computations. Using them actions implementing domain logic can be chained together and decorated with custom code. C++ template metaprogramming could benefit from adopting monads in situations like advanced error propagation and parser construction. In this paper we present an approach for implementing monads in C++ template metaprograms. Based on this approach we have built a monadic framework for C++ template metaprogramming. As real world examples we present a generic error propagation solution for C++ template metaprograms and a technique for building compile-time parser generators. All solutions presented in this paper are implemented and available as an open source library.

Keywords: C++ template metaprogram, monad, exception handling, monoid, typeclass
2010 MSC: 68N15, 68N18

1. INTRODUCTION

Templates are key elements of the C++ programming language [33], capturing commonalities of abstractions without performance penalties at runtime. In 1994 Erwin Unruh wrote a C++ program [34] which didn't compile, however, the error messages emitted by the compiler displayed a list of prime numbers. Unruh used C++ templates and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [38]. These compile-time programs are called *C++ template metaprograms* and form a Turing-complete sub language of C++ [5].

Today programmers write metaprograms for various reasons, like implementing *expression templates* [36], where runtime computations can be fine-tuned with compile-time activities to enhance runtime performance; *static interface checking*, which increases the ability of the compiler to check the requirements against template parameters by specifying constraints on them [15, 26]; *active libraries* [37], acting dynamically at compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [6] for type-safe SQL interface, Boost.Xpressive [41] for regular expressions or Metaparse [23] for parsing at compile-time.

Initially C++ template metaprograms were constructed in an ad-hoc way, which led to serious maintenance problems. Krzysztof Czarnecki and Ulrich Eisenecker introduced the idea of looking at the building blocks of template metaprograms – template classes – as functions evaluated at compile-time [5]. The arguments of the functions are the template parameters of the classes, the values of the functions are nested types of the template classes. The execution of a metaprogram is the evaluation of a template metafunction. That metafunction can call other template metafunctions – that is, trigger the instantiation of other template classes. The instantiation of a template class cannot change any global state in the compilation process and instantiating the same template with the same arguments always gives the same result, thus template metafunctions are *pure function*s.

One can look at template metaprogramming as a pure functional language [5]. Based on this similarity, a number of useful techniques for functional languages, such as Haskell [21], can be adopted in C++ template metaprogramming.

In Haskell, a monad [21] is a programming structure representing some form of a computation that can be used to chain a number of functions together. A monad can decorate the functions that are chained together to implement some general logic that is orthogonal to the chained functions. An example of such orthogonal logic is error propagation in a sequence of functions – when one of them fails and returns an error, the error should be returned to the caller without evaluating the rest of the functions. Monads have various use cases nowdays. A few examples:

- Input and output is implemented using a special monad, called the *IO monad* [21] in Haskell.
- Monads can help the implementation of parser combinators. Parsers can be combined in a monadic way and the monadic framework can take care of a large amount of boilerplate code during this process.
- Monads can be used to simplify error propagation in complex code. By using them, the error propagation logic can be separated from the business logic. They can be used to replace exceptions in pure code.
- Monads can simplify the implementation of pure code operating on a state. Different types of monads can handle mutable and immutable states.
- Monads can simplify the implementation of pure code doing logging or producing other type of output. A monad can take care of collecting that output.

When we are following the object oriented programming paradigm, we can abstract either the data a computation is working on or the computation that works on a piece of data. To abstract the data a piece of code is working on one can use the Composite design pattern [10]. When one uses that, a computation doesn't need to know the real type of the object it is working on. To abstract the computation, one can use the Command design pattern [10]. Using it, one can abstract pieces of a larger computation away. The rest of the code can be implemented in a way that is not aware of what these pieces are doing.

In functional programming monads are a tool for abstracting computation. Users of a monad combine different functions together using the operations provided by the monad. The code combining these pieces together is generic, it is implemented without knowing what these pieces will be.

A monad operates on functions that are passed around as values taking advantage of the fact that functions are first class citizens in functional programming. The Command design pattern uses inheritance and runtime polymorphism to be able to pass small pieces of functionality around as values [21].

Given the similarities of Haskell and C++ template metaprogramming, the idea of monads can be implemented in C++ template metaprogramming as well. In this paper we present an approach for porting Haskell code to C++ template metaprogramming and how it can be used to implement monads. Using this technique we port a number of monads from Haskell to template metaprogramming.

The *do syntax* [21] improves the readability of monadic Haskell code. This is a syntactic sugar for code using monads. We present how this syntactic sugar can be implemented in C++ template metaprogramming to improve the readability of template metaprograms.

We present two real world use cases of monads in template metaprogramming. One of them is an approach for implementing error propagation in pure code and an embedded language for C++ template metaprogramming that resembles the syntax of exception handling in runtime code. The other example we present is a monadic extension of the compile-time parser generator library we presented in an earlier paper [23]. The library was designed in a monadic way, however it didn't use any framework supporting monads. We present how using one simplifies code using the library. All solutions presented in this paper are implemented and available as an open source library [44].

The rest of the paper is organized as follows. We present the C++ template metaprogramming implementation of algebraic data types and type classes in Section 2 and monads in Section 3. A number of well-known Haskell monads and their C++ template metaprogramming implementations are discussed in Section 4. We present how a syntactic sugar, the *do notation* can be implemented in C++ template metaprogramming in Section 5. Two real world examples are presented in Section 6. Related research results are discussed in Section 7. Our paper concludes in Section 8.

2. PREREQUISITES

Haskell monads are implemented using algebraic data types and typeclasses. To be able to implement monads, we need a way of transforming them to C++ template metaprogramming. This section presents these language elements and the way they can be transformed.

2.1. ALGEBRAIC DATA TYPES

We use the approach presented in an earlier paper [23] for representing Haskell's algebraic data types in C++ template metaprogramming. Algebraic data types in Haskell have the following form:

```
data <name> [<type arguments>] =
  <constructor name> <constructor arguments> |
  <constructor name> <constructor arguments> |
   ...
```

We implement each constructor by a C++ template. The constructor arguments are the template arguments. For example the constructor `Div Expr Expr` is implemented as

```
template <class Expr1, class Expr2> struct div
{ typedef div type; /* Needed for lazy evaluation */ };
```

We couldn't express Haskell types in C++ template metaprograms, the type of the template arguments is always `class`. Algebraic data types and their arguments have no direct representation in C++ template metaprogramming, only the constructors are implemented.

2.2. TYPECLASSES

The Haskell language provides typeclasses [21] for implementing function overloading. There is a known similarity between Haskell typeclasses and C++ concepts [4], however, concepts have been removed from the new standard. We present a solution for implementing typeclasses in conformity with the C++ standard.

A typeclass defines an interface for a type. It takes the type as argument and declares a number of functions using that type in their signature. The following example shows the syntax of creating a typeclass:

```
class EqualityComparable a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
```

This example defines a typeclass called `EqualityComparable`. Its argument is called `a` and two functions are specified: `equal` and `notEqual`. Both of them take two values of type `a` as arguments and return a boolean value.

Types can be instances of a typeclass. Every type has to be explicitly made an instance of a typeclass by implementing the expected functions.

The following example shows the syntax of making a type an instance of a typeclass:

```
instance EqualityComparable Int where
  equal x y = x == y
  notEqual x y = x /= y
```

This example uses the comparison operators for implementing the two functions. Certain functions required by a typeclass can have default implementations. Instances can override this default, but every instance not overriding it inherits the default version. The following example shows the syntax of providing a default implementation for a function:

```
class EqualityComparable a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
  notEqual x y = not (equal x y)
```

This example uses `equal` to implement `notEqual`.

Typeclasses can be implemented in C++ template metaprogramming based on the idea of *traits* [20]. A trait is a template class with member types and static member constants. This template class can be specialised for different types as template arguments and define the nested types and static member constants differently for every template argument. It is used to encode extra information about types that can be consumed by template metaprograms.

A typeclass can be implemented as a trait. The argument of the typeclass is the template argument of the trait. The list of expected functions cannot be explicitly encoded. The following example shows the `EqualityComparable` typeclass implemented in template metaprogramming:

```
template <class A> struct equality_comparable;
```

This template class has no implementation. This ensures that when it is used inappropriately, the compiler emits an error message at the moment of misuse and the user doesn't get a confusing error message at a later point in the compilation process.

Boost.MPL [40] uses *tag*s to implement template metafunction overloading [1]. A tag is a class that is used as an identifier in template metaprogramming. Boost.MPL uses tags as dynamic type information. Our implementation of typeclasses expects tags as template arguments.

A tag can be made an instance of a typeclass by specialising the template for that tag and implementing the expected functions as template metafunction classes – classes with a nested metafunction called `apply` [1]. The following example shows how to make the boxed integers of Boost.MPL instances of our example typeclass.

```
template<> struct equality_comparable<integral_c_tag> {
  struct equal {
    template <class A, class B>
    struct apply : boost::mpl::equal_to<A, B> {};
  };
  struct not_equal {
    template <class A, class B>
    struct apply : boost::mpl::not_equal_to<A, B> {};
  };
};
```

This code specialises the `equality_comparable` template class for the type `integral_c_tag` and implements the two expected operations, `equal` and `not_equal`, as metafunction classes. These implementations use comparison functions provided by Boost.MPL.

The trait implementing the typeclass can be used to call functions related to a typeclass. Unfortunately the calling code has to specify the tag explicitly. The following example implements a function, `self_equal`, using the `equality_comparable` typeclass in both languages:

```
-- Haskell
selfEqual :: EqualityComparable a => a -> Bool
selfEqual x = equal x x
```

```
// Template metaprogramming
template <class X> struct self_equal : apply<
  typename equality_comparable<
    typename boost::mpl::tag<X>::type>::equal,
  X
> {};
```

The requirement, that the argument `x` has to be an instance of a typeclass is encoded in a different way in the two languages. In Haskell it is encoded in the type of the function by having an expectation on the type argument, while in template metaprogramming it is encoded in the implementation of the function by accessing an element of the trait.

Expected functions with default implementations can be implemented in template metaprogramming as well by creating a second template class for the typeclass containing the default implementations as metafunction classes. Every instance of the typeclass has to instantiate this extra template class and inherit publicly from the instance. Here is an example for the extra template class and the updated instance:

```
template<class A> struct equality_comparable_defaults {
  struct not_equal {
    template <class A, class B> struct apply :
      boost::mpl::not_<typename boost::mpl::apply<
        typename equality_comparable<A>::equal, A, B
      >::type> {};
  }; };
template<> struct equality_comparable<integral_c_tag> :
equality_comparable_defaults<integral_c_tag> {
  // not_equal is inherited
  struct equal { /* Same as before... */ }; };
```

The default implementation of `not_equal` uses the `equal` method of the `equality_comparable` typeclass. Any instance can override this default implementation by overriding the nested class. Using this approach to implement typeclasses in template metaprogramming has several advantages.

- Helps structuring the code by collecting the functions implementing the same abstract concept – what the typeclass represents – together in one class.

- Given the fact that typeclasses are always used explicitly, it helps the compiler provide meaningful error messages, since the name of the typeclass is likely to appear in the error messages when a tag is not an instance of the typeclass the code is trying to use.

This approach has several drawbacks as well.

- It doesn't support specifying the list of expected functions. The author of a typeclass can express it using comments or in the documentation, but not in a way that the compiler understands.

- The compiler can not verify and enforce the existence and the expected signature of the required functions. Error messages are generated the first time a missing function is called.

In spite of the drawbacks, following this approach helps making template metaprograms more structured.

3. MONAD IMPLEMENTATION

In Haskell a monad is implemented by a typeclass, called `Monad`. It takes a type constructor as argument. The type constructor has to take one argument to produce a type. Instances of `Monad` are called *monadic types*, values of those types are called *monadic values*. The typeclass requires the following operations to be implemented:

```
class Monad m where
  return :: a -> m a
  fail :: String -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b

  a >> b = a >>= \_ -> b
  fail = error
```

Two operators are required, both of them taking two arguments:

- `>>=`, taking a monadic value and a function mapping a value of some type to a monadic value. The operator builds a new monadic value. This operator can decorate the function or decide not to evaluate it at all. Since the first argument of this operator can be the result of another call of this operator, it can be used to chain a number of functions mapping values to monadic values together. This operator can implement some general logic that is orthogonal to what the functions that are chained together do.

- `>>`, taking two monadic values and building a new one. The typeclass provides a default implementation for this function that calls the `>>=` operator with a function that always returns the second argument of `>>`.

As an addition, two functions are required:

- `return`, taking a value of some type and returning a monadic value. The purpose of this function is to lift a value into the monad.
- `fail`, taking a string and returning a monadic value. The returned value has to break a chain of functions built with the `>>=` operator. This function has a default implementation that calls `error`, which generates an exception [21]

As an example for monads we can think of a simple way of error handling. A function returning type `a` can return a type `Maybe a` instead. A `Maybe a` value can be either `Just a`, which means that there was no error or `Nothing` which means that something went wrong. The definition of `Maybe` in Haskell looks like the following:

```
data Maybe a = Just a | Nothing
```

An instance of monad can be defined for `Maybe`. The `Maybe` monad keeps calling the functions in a chain as long as they succeed. Any error breaks the chain. The expected operations can be defined the following way:

```
instance Monad Maybe where
  return x = Just x

  Nothing  >>= _  = Nothing
  (Just x) >>= f  = f x
```

`return` wraps its argument to represent a successful result. The bind operation returns `Nothing` if the result of the previous operation failed and applies the next operation otherwise. This was just one example for a monad. We will present further examples later.

In C++ template metaprogramming we represent the monadic values as a set of template metaprogramming values. Since template metaprogramming is weakly typed, this set can be defined in an informal way – in a comment or in the documentation. In Haskell the compiler can verify if a value is monadic or not, while the C++ compiler can not do it for us in template metaprogramming. On the other hand, this makes the definition of monadic values more flexible – as we will see, there are sets of monadic values that can be defined in template metaprogramming but not in Haskell. Monads can be implemented using typeclasses requiring the following metafunctions:

- `return_`, taking some metaprogramming value as argument and returning a monadic value. This is the equivalent of `return`.

- `bind`, taking a monadic value and a metafunction class as arguments and returning a monadic value. The metafunction class takes some value and returns a monadic value. This is the equivalent of the `>>=` operator.
- `bind_`, taking two monadic values as arguments and returning a new monadic value. This metafunction evaluates its arguments lazily [27], thus its arguments can be nullary metafunctions returning a monadic value. It can replace `bind` in cases where the metafunction class ignores its argument. This function is implemented by the `>>` operator in Haskell.
- `fail`, taking some value as argument and returning a monadic value. Its purpose is reporting errors in monads. When it is used in a chain of `bind` calls, the returned value should break the evaluation of the chain. This function is called `fail` in Haskell as well.

Since operators can not be used in template metaprogramming, we had to give the operations names. We create a typeclass called `monad` as the equivalent of the `Monad` typeclass in Haskell:

```
template <class Tag> struct monad;
  // Requires:
  //    return_::apply<T>, fail::apply<T>
  //    bind::apply<T, F>, bind_::apply<T, V>
```

As `>>` and `fail` have default implementations in Haskell, we can provide a default implementation for `bind_` and `fail` in template metaprogramming as well:

```
template <class T> struct monadic_error {};

template <class Tag> struct monad_defaults {
  struct bind_ {
    template <class A, class B> struct apply :
      boost::mpl::apply_wrap2<
        typename monad<Tag>::bind,
        A, boost::mpl::always<B>>
    {};
  };
  struct fail {
    template <class T> struct apply :
      monadic_error<T>::failed {};
  };
};
```

bind_'s default implementation calls bind with a metafunction class that always returns bind_'s second argument. In Haskell fail's default implementation uses error, which is something we do not have in C++ template metaprogramming. However, we can replace it with a code that breaks the compilation process by accessing a non-existing nested type in a template class. The name of the nested class is likely to appear in the error message generated by the compiler, thus by giving this class a meaningful name we can improve the quality of the error message a bit. The example above uses a non-existing nested class called failed. Since we're trying to access a nested class in a template class instance, it doesn't generate any error message until it is instantiated. Every instance of monad has to publicly inherit from monad_defaults to get the default implementations.

To simplify using monads, we can create wrapper template metafunctions for the functions expected by the monad. The tag of the monad has to be the first argument of these metafunctions.

```
template <class Tag, class T> struct return_ :
  boost::mpl::apply<typename monad<Tag>::return_,T> {};
```

The above example shows how a helper function for return_ can be implemented. The rest of the functions (bind, bind_ and fail) can be implemented in a similar way.

Haskell has semantic expectations for monads [21] that are documented but cannot be verified by the compiler. The C++ template metaprogramming equivalent of these expectations are the following:

- *left identity*: bind<Tag, return_<Tag, X>, F> is equivalent to boost::mpl::apply<F, X>.
- *right identity*: bind<Tag,M,monad<Tag>::return_> is equivalent to M.
- *associativity*: The expression bind<Tag, M, lambda<x, bind<Tag, boost::mpl::apply<F, x>, G>>> is equivalent to bind<Tag, bind<Tag,M,F>,G>. lambda is our lambda expression implementation [30].

Similarly to Haskell, we cannot verify these expectations. It is the responsibility of the monad's author to satisfy these expectations.

4. MONAD VARIATIONS

In this section we present how different types of monads available in Haskell can be implemented in C++ template metaprogramming. The full implementation of these monads is part of Mpllibs [44].

4.1. MAYBE

`Maybe` has the following definition in Haskell:

```
data Maybe a = Nothing | Just a
```

It can be used as a basic error handling mechanism: a function either returns some result (`Just a`) or a special value representing error (`Nothing`). `Maybe` is a monad instance, `return` wraps its argument with `Just`, `fail` returns `Nothing`, `bind` implements error propagation logic: it stops evaluating the chained functions when one of them returns `Nothing`.

The `Maybe` type can not be implemented as a type in template metaprogramming, only as a set of individual data-constructors. Template metaprogramming can not express the connection between them, we need to do that in the documentation. `Nothing` can be implemented by an empty class:

```
struct nothing {};
```

`Just` can be implemented by a template class:

```
template <class A> struct just {};
```

We need to create a trivial metafunction, `is_nothing`, checking if a value is `nothing` or not – we do not present it here. We need a new tag representing the `Maybe` monad:

```
struct maybe {};
```

Having all these elements we can express that `Maybe` is an instance of the `monad` typeclass:

```
template <> struct monad<maybe>: monad_defaults<maybe> {
  struct return_
   { template <class T> struct apply : just<T> {}; };
  struct fail
   { template <class S> struct apply : nothing {}; };

  struct bind {
    template <class A, class F> struct call_F :
      boost::mpl::apply<F, typename get_data<A>::type>
    {};

    template <class A, class F> struct apply :
      boost::mpl::if_<is_nothing<A>,
        boost::mpl::identity<A>, call_F<A,F>>::type
    {};
}; };
```

`return_` wraps its argument with `just`, `bind` checks if its first argument, the result of the previous step in the chain is `nothing`. When it is, it returns this value without calling the next step. Otherwise it unwraps the value from `just` and passes it to the next step in the chain. `fail` returns `nothing` to break the chain of `bind`s.

This monad implements some error propagation logic. It can be used to combine metafunctions using `Maybe` to report errors. The problem with this solution is that the monadic functions can not return any detail about the error.

4.2. EITHER

The `Either` monad can be used for error handling as well. In Haskell the following type is defined:

```
data Either a b = Left a | Right b
```

When it is used for error handling, `Left a` represents an error, `Right b` represents a result. Since `Left` has an argument as well, functions using `Either` for error reporting can report details describing what went wrong. `Either` is a monad instance as well, `bind` implements error propagation logic. The type constructors can be implemented as template classes in C++ template metaprogramming:

```
template <class A> struct left {};
template <class B> struct right {};
```

Similarly to Maybe, we need an is_left metafunction. Its implementation is trivial, we do not present it here. We need to create a new tag, either, for the Either monad. We can make either an instance of monad:

```
template<> struct monad<either>:monad_defaults<either>{
  struct return_
   { template <class T> struct apply : right<T> {}; };
  struct fail
   { template <class S> struct apply : left<S> {}; };

  struct bind {
    template <class A, class F> struct call_F :
      boost::mpl::apply<F, typename get_data<A>::type>
    {};

    template <class A, class F> struct apply :
      boost::mpl::if_<is_left<A>,
        boost::mpl::identity<A>, call_F<A, F>>::type
    {};
}; };
```

return_ wraps its argument with right to make it a result, fail wraps its argument with left to break the evaluation of the monad. bind propagates the error, when its first argument is left. When its first argument is right, it unwraps the value and calls the monadic function.

Using this monad a better error-handling logic can be implemented since monadic functions can return information about what the problem was in case of errors.

4.3. LIST

The list monad turns operations mapping elements to lists into operations transforming lists. Monadic values are lists of some type. return_ creates a list with one element. bind's first argument is a list. It calls the monadic function on all elements of this list and concatenates the resulting lists. Since the List monad doesn't deal with error handling, there is no reasonable way of overriding fail. It can be implemented the following way:

```
struct join_lists {
  template <class State, class NewList> struct apply :
    boost::mpl::insert_range<State,
      typename boost::mpl::end<State>::type, NewList>
  {};
};

template <> struct monad<list_tag> :
  monad_defaults<list_tag> {
  struct return_ {
    template<class T>
    struct apply : boost::mpl::list<T> {};
  };
  struct bind {
    template <class A, class F>
    struct apply : boost::mpl::fold<
        typename boost::mpl::transform<A, F>::type,
        boost::mpl::list<>, boost::mpl::join_lists>
    {};
  };
};
```

The `return_` operation builds a one element list from its argument. The `bind` operation applies its second argument, the function on all elements of the list, which is its first argument. This application is implemented using the `transform` metafunction provided by Boost.MPL. The result of this is a list of lists, which need to be concatenated. This happens by folding over this list using a helper metafunction, `join_lists`. This metafunction joins its two arguments using `boost::mpl::insert_range`. The list monad, which we get by this implementation can be used to implement ambiguity in pure code [11, 12].

4.4. READER

The reader monad combines functions operating on an immutable state. Monadic values are higher order functions taking the state as argument and returning some value. The monad itself doesn't deal with the state – it constructs functions operating on it. The result of a chain of `bind`s is a function that takes the state as its argument.

In C++ template metaprogramming higher order functions are implemented using metafunction classes, thus in the template metaprogramming Reader monad monadic values are metafunction classes. A tag, `reader` needs to be created to make Reader an instance of `monad`:

```
template<> struct monad<reader>: monad_defaults<reader>{
  struct return_ {
    template <class T> struct apply
      { typedef boost::mpl::always<T> type; };
  };
  struct bind {
    template <class A, class F> struct impl {
      template <class R> struct apply :
        boost::mpl::apply<typename boost::mpl::apply<
          F, typename boost::mpl::apply<A, R>::type
        >::type, R> {};
    };
    template <class A, class F>
    struct apply : impl<A, F> {};
  };
};
```

`return_` creates a constant function – regardless of the state it always returns `return_`'s argument. The function created by `bind` takes a state as argument and passes it to `bind`'s first argument. The resulting value is used to construct a new `state -> value` function. The state is passed to this function to get the final result.

In the reader monad, monadic functions construct functions operating on the state based on the result of the previous function operating on the state. Thus, the execution of higher order code – code building functions operating on the state – is mixed with normal functions operating on the state.

4.5. STATE

The State monad maintains a state like the Reader monad, but the monadic values are functions that can change the state: they are functions taking a state as an argument and returning a pair: a new state and a result.

The C++ template metaprogramming implementation of this monad is similar to the implementation of the Reader monad: higher order functions are represented by metafunction classes, pairs are implemented using pairs provided by Boost.MPL.

```
template<> struct monad<state>: monad_defaults<state> {
  struct return_ {
    template <class T> struct apply { struct type {
      template <class S>
      struct apply : boost::mpl::pair<T, S> {};
    }; };
  };

  struct bind {
    template <class A, class F> struct impl {
      template <class S> class apply :
        boost::mpl::apply<
          typename boost::mpl::apply<
            F, typename apply_first::first>::type,
          typename
            boost::mpl::apply<A, S>::type::second
        > {};
    };
    template <class A, class F>
    struct apply : impl<A, F> {};
  };
};
```

`return_` creates a function returning `return_`'s argument and not changing the state. The function created by `bind` takes a state as argument and passes it to `bind`'s first argument. The resulting value is used to construct a new `state -> (value, state)` function. The new state is passed to this function to get the final result.

Given that C++ template metaprogramming is a pure functional language, there is no mutable global state. However, using the State monad functions having to operate on a mutable global state can be implemented by simulating a state using the monad in C++ template metaprograms. An example use of it is building parsers in a monadic way, which we discuss in detail in section 6.4.

The Writer monad demonstrates the expressiveness of our typeclass implementation and an extension to tags used by Boost.MPL. To be able to implement the Writer monad, we need to implement monoids. In abstract algebra an object is called a monoid [21] if it meets the following requirements:

- It has an associative binary operator. That is, an operator, `*`, that satisfies the following equation: `a * (b * c) == (a * b) * c`.
- It has an identity value, `e`, that satisfies `a * e == a` and `e * a == a`

In Haskell, this concept is captured by the `Monoid` typeclass:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

`mappend` implements the binary operation, `mempty` is the identity element. `mconcat` is a function concatenating the elements of a list using the binary operation. It has a default implementation that can be overridden by a more efficient algorithm for types where it is possible. This typeclass can be implemented in template metaprogramming using the approach we have presented for implementing typeclasses. The full implementation can be found in Mpllibs [44]. The monadic values of the Writer monad are pairs: a value and a state. The states are expected to form a monoid [21], thus they have an associative operation that can be used to merge a number of state values. The Writer monad collects the list of states while executing a chain of `bind` calls and reduces them into one value using the reduction function of the monoid.

Creating a tag for the Writer monad is not as straight forward as it was for other monads, since the Writer monad expects a monoid instance as argument. The Haskell implementation expects the type of the state to be an instance of the `Monoid` typeclass. We need to provide an extra argument to the Writer monad: the tag of the monoid. It can be provided by making the tag of the Writer monad a template class taking the tag of the monoid as argument.

```
template <class Monoid> struct writer {};
```

`writer` can be an instance of the `monad` typeclass using partial specialisation [35]:

```
template <class Monoid> struct monad<writer<Monoid>> :
  monad_defaults<writer<Monoid>> { /* ... */ };
```

It makes `writer` an instance of the `monad` typeclass independent of the monoid the Writer monad uses. Using our typeclass implementation, the functions expected by `monad` can be implemented in a generic way, without needing to know which monoid is used:

```
template <class Monoid> struct monad<writer<Monoid>> :
 monad_defaults<writer<Monoid>> {
  struct return_ { template <class T> struct apply {
    typedef boost::mpl::pair<
      T, typename monoid<Monoid>::empty> type;
  }; };

  struct bind{ template <class A, class F> struct apply{
    typedef typename
      boost::mpl::apply<F, typename A::first>::type FA;

    typedef boost::mpl::pair<
      typename FA::first ,
      typename boost::mpl::apply<
        typename monoid<Monoid>::append ,
        typename A::second , typename FA::second
      >::type> type;
  }; };
};
```

This code uses the `monoid` typeclass to refer to the monoid's operations. Since the `Monoid` argument refers to a tag of a monoid, the nested types `empty` and `append` are expected to be defined in the `monoid<Monoid>` trait instance.

Our typeclass implementation allowed us to implement the Writer monad independently of the monoid used by the monad. We have encoded the tag of the monoid in the tag of the Writer monad.

5. DO NOTATION

Haskell provides syntactic sugar for monads, called *do notation* [21]. In Haskell, a *do block* is associated with a monad and contains a number of monadic function calls and value bindings. Here is an example do block:

```
do
  r <- may_fail1 13
  may_fail2 r
```

This evaluates `may_fail1 13`, binds `r` to its result and evaluates `may_fail2 r`, thus it does the same as we did in our previous example but is easier to read. When the code gets more complicated, this difference becomes more significant. We propose the following solution for supporting do notation in C++ template metaprogramming. A do block looks like the following:

```
do_<monad_tag>::apply<
  step1,
  // ...
  stepn>
```

`do_` is a template class, `monad_tag` is the `tag` identifying the monad. This has to be passed to the `return_` and `bind` functions. `do_<monad_tag>` is a metafunction class taking the steps of the do block as arguments. A step is either a nullary metafunction returning a monadic value or a binding of an expression to a name. A binding is expressed by the following structure:

```
set<name, step>
```

`step` is a nullary metafunction returning a monadic value, `name` is the name of a class. The binding binds the result of `step` to this name. The bound name can be used in the steps of the do block following the binding. Our example using the `may_fail` functions can be written the following way:

```
struct r;
do_<exception_tag>::apply<
  set<r, may_fail1<boost::mpl::int_<13>>>,
  may_fail2<r>>
```

This implementation uses `set` to bind the result of calling the `may_fail1` metafunction to a name, `r`. This result can be passed to `may_fail2` using the name `r`.

5.1. Implementation of the do notation

We followed the approach presented in [21] to implement the desugaring of do blocks. It uses a special version of `bind` that takes two monadic values as arguments and returns a new monadic value. It can be used to implement steps in the sequence of monadic function calls where functions ignore the result of the previous function. We implemented this special `bind` as a template metafunction called `bind_`. According to [21] it can be implemented in terms of `bind`:

```
template <class MonadTag, class A, class B>
struct bind_ :
  bind<MonadTag, A, boost::mpl::always<B>> {};
```

This implementation passes a special metafunction class always returning `B` as the monadic function argument of `bind`. Our real implementation is similar to this one, we do not present it here in detail. It can be found in [44].

In our implementation `do_<monad_tag>::apply<step1, ..., stepn>` instantiates `do_impl<monad_tag, step1, ..., stepn>`, which evaluates the do block itself. This transition can be implemented using the Preprocessor metaprogramming library of Boost [39]. Our implementation can be found in [44]. `do_impl` can be implemented the following way:

- `do_impl<monad_tag, step>` evaluates `step`.
- `do_impl<monad_tag, step1, step2, ..., stepn>` evaluates `bind_< monad_tag, step1, do_impl<monad_tag, step2, ..., stepn>`.
- `do_impl<monad_tag, set<name, exp>, step2, ..., stepn>` evaluates `bind<monad_tag, exp, lambda<name, do_impl<step2, ..., stepn> >>`. We use our own lambda expression library which can be found in [44]. `lambda<name, nullary_metafunction>` takes a class, `name`, and a nullary metafunction as arguments. `lambda` is a metafunction class that takes a class as an argument. It constructs an updated version of `nullary_metafunction`, in which all occurrences of `name` are replaced with the value of `lambda`'s argument. This updated nullary metafunction is evaluated and `lambda` returns the result of this evaluation. Further information on how it works can be found in [28].

Following these rules, the example:

```
struct r;

do_<exception_tag>::apply<
  set<r, may_fail1<boost::mpl::int_<13>>>,
  may_fail2<r>>
```

is transformed into

```
struct r;

bind<
  exception_tag,
  may_fail1<boost::mpl::int_<13>>,
  lambda<r, may_fail2<r>>>
```

When a do block is evaluated, it is transformed to a nullary metafunction like the example above and gets evaluated. The transformation happens when the do block is evaluated, thus do blocks that are not evaluated are never transformed.

5.2. Using return_ in do blocks

The standard tool for creating monadic values from non-monadic ones is `return_`. It takes the `tag` of the monad and the non-monadic value and returns a monadic value. It can be used in do blocks the following way:

```
struct r; struct s;

do_<monad_tag>::apply<
  set<r, return_<monad_tag, boost::mpl::int_<11>>>,
  set<s, return_<monad_tag, boost::mpl::int_<13>>>,
  call_some_function<r, s>>
```

The problem with this solution is that, every time we use `return_` inside the do block, we need to pass the `tag` of the monad to it, as well. It requires specifying the `tag` of the monad at multiple places, which is more work to be done for the developer and more possibilities to make mistakes, leading to bugs that are difficult to find. We have implemented a solution that makes the do block deduce the first argument of `return_`. Developers can use `do_return`, a template class inside do blocks. Here is the previous example using this new tool:

```
struct r; struct s;

do_<monad_tag>::apply<
  set<r, do_return<boost::mpl::int_<13>>>,
  set<s, do_return<boost::mpl::int_<14>>>,
  call_some_function<r, s>>
```

`do_return<x>` is substituted with `return_<monad_tag, x>` inside a do block. It can be implemented in a similar way as let and lambda expressions. The implementation has to take care of cases when `do_` blocks are nested in each other. This problem is similar to let expressions containing other let expressions and can be solved using the same approach. The details of implementing let and lambda expressions are presented in [28]. With this, the implementation of do blocks is complete.

As in Haskell, using do blocks makes the source code easier to read and understand in C++ template metaprograms as well.

6. USE CASES OF MONADS

Using monads in C++ template metaprogramming has several benefits. In this section we present two real world use cases of monads – how they can be implemented and what value they add to template metaprogramming.

6.1. COMPILE-TIME EXCEPTION HANDLING

We have presented two monads, Maybe and Either, targeting error handling in pure code. In this section we present a third monad, which we call the Exception monad, that can handle errors in C++ template metaprograms. We present how this monad can be extended to simulate exception handling in C++ template metaprograms.

Our Exception monad treats every value in C++ template metaprogramming as a monadic value. Note that this wouldn't be possible in Haskell, since that language uses the type system to define the monadic values. We can create a special data-constructor for representing errors:

```
template <class Detail> struct exception {};
```

`exception` values contain details about the error, this is what the `Detail` argument represents. The exception monad follows the same logic as the Either monad, treating `exception` values as `left` and other values as `right` values. Thus, `return_` is the identity function, `bind` implements error propagation.

The Exception monad stops the further execution of the chain of binds in case of an exception and propagates the error to the caller, who can either process this error information or propagate it further. This behaviour is the same as exceptions have at runtime. [33]

We present compile-time exception handling using the `min` template metafunction as an example: it takes two arguments and returns the smaller one. It uses another metafunction, `less`, to decide which is the smaller argument. `min` can be implemented (and is implemented in Boost.MPL) the following way:

```
template <class A, class B>
struct min : if_<less<A, B>, A, B> {};
```

When `less` returns an exception, the first argument of `if_` is an exception instead of a logical value. The body of `min` is a template metaprogramming expression. A sub-expression of it, `less<A, B>`, calls another metafunction, `less`. When a sub-expression of an expression returns an exception, the exception propagation logic should stop the evaluation of the entire expression and make the exception the result of the expression (thus, propagate the exception). In order to do this, we have to turn the above example into monadic code:

```
struct t;

template <class A, class B> struct min :
  bind<exception,
    less<A, B>, lambda<t, if_<t, A, B>>> {};
```

`lambda` is our lambda-expression implementation presented in [29]. `t` is the argument of the lambda-expression, `if_<t, A, B>` is the body of it. The code evaluates the original expression in two steps: first it evaluates the sub-expression that may return an exception, then it evaluates the rest of the expression. The two steps are connected by `bind`.

Turning every template-metaprogramming expression into monadic code is a tedious and error prone process. It makes the code extremely difficult to read and maintain. We present a way to implement a small embedded language for C++ template metaprogramming that resembles runtime exception handling. This language allows the developer to use *try* and *catch* blocks in template metaprograms. These blocks are automatically translated into monadic code presented above. The embedded language we present can be implemented using the C++ standard, it doesn't require any additional tools.

We introduce the *compile-time try* block, which is a template class taking one argument: a nullary metafunction. The try block turns the nullary metafunction into a monadic expression before evaluating it. Using it `min` can be implemented the following way:

```
template <class A, class B>
struct min :
  try_<if_<less<A,B>, A,B>> {};
```

This solution wraps the body of the original `min` implementation. `try_` is a template class taking a nullary metafunction as argument. It transforms this nullary metafunction into a series of `bind_` calls:

- When the nullary metafunction is a class that is not a template instance, it remains as it is.
- When the nullary metafunction is an instance of a template class, it is transformed into a series of `bind_` calls. An instance of the `f` template class with `T1` ... `Tn` arguments, `f<T1, ..., Tn>`, is transformed into the following:

```
struct t1; /* ... */ struct tn;

bind<exception, T1, lambda<t1,
 bind<exception, T2, lambda<t2, /* ... */
  bind<exception, Tn, lambda<tn, f<t1,/*...*/,tn>>>
 /* ... */ >>
>>
```

This transformation ensures that when a sub-expression of the nullary metafunction throws an exception, the exception is not passed to the function taking that value as an argument but is propagated out of the entire expression. Using these try blocks the exceptions can be propagated in the chain of function calls, similarly to stack unwinding in runtime code. This transformation is similar to the logic of desugaring *do blocks* in Haskell [21]

Instances of the `try_` template provide a nested type called `type`, that is a `typedef` of the result of the monadic calculation. Thus, `try_` can be used as a metafunction. As it is the case in runtime code, exceptions are either handled at some point or they are propagated out of the entire metaprogram and break the evaluation of it. In runtime code they can be handled using *catch* blocks. Catch blocks can filter the exceptions by type and catch exceptions of a certain type or its subtypes only. Another option is to catch every exception regardless of its type. A try block is followed by any number – including zero – of catch blocks. When any of the catch blocks handles the exception, the execution of the program continues after the try block. When none of the catch blocks catches the exception, it is propagated further.

When a metafunction needs to handle exceptions propagated out of an expression, the metafunction has to check the result of that expression. The following code calls `min` and handles any exceptions thrown by `min`:

```
template <class N> struct max_zero : eval_if<
  typename is_exception<min<N, int_<0>>>::type,
  /* error handling code goes here */, min<N, int_<0>>
> {};
```

This code uses a metafunction, `is_exception` to check if `min` returned an exception or not. When it did, an error handling branch of an `eval_if` is called, otherwise the result of `min` is returned. When the error handling code has to differentiate different types of exceptions, it needs a chain of nested `eval_if`s to detect the type of the exception.

Our embedded language for exception handling can be extended to support the developers of template metaprograms. We introduce the idea of the *compile-time catch block*, which is a template metafunction class taking one argument: the error handling code as a nullary metafunction. The enclosing class of the metafunction class is a template class taking two classes as template arguments:

- A tag, which is used as a filter: the catch block catches an exception, when the data of it – the value that was *thrown* – has the same tag. Our embedded language has a special tag, `catch_any`, that catches every exception.
- A place holder class. In the error handling nullary metafunction every occurrence of the place holder class is replaced by the data of the exception that was thrown.

Here is an example catch block:

```
struct e; // Placeholder class
struct range_error_tag; // Tag: out-of-range exceptions

// Metafunction getting the last
// valid element of the range
template <class RangeError> struct get_range_boundary;

catch_<range_error_tag, e>
  ::apply<get_range_boundary<e>>
```

The above example shows a catch block that returns the last valid element of a range in case of an out of range exception. Catch blocks belong to try blocks, thus catch blocks are implemented as nested template classes of try blocks. Using them `max_zero` can be implemented the following way:

```
struct comparison_error_tag;

template <class N> struct max_zero :
  try_<min<N, int_<0>>>
    ::template catch_<comparison_error_tag, e>
      ::template apply< /* handle comparison error */ >
    ::template catch_<catch_any, e>
      ::template apply< /* handle other errors */ >
{};
```

The catch blocks operate on the result of the try block: they catch it when they can catch it according to the tag of the error. The evaluation logic of a catch block is the following:

- When there was no exception or the catch block can not catch it according to its tag, the catch block returns the result of the try block and ignores the error handling nullary metafunction.
- When the catch block can catch it and no previous catch block has caught it, the catch block evaluates the error handling nullary metafunction.
- The result of a catch block contains a nested `catch_` template to make chaining catch blocks possible.

Using this logic compile-time catch blocks have the same logic as the runtime ones: they check the exceptions in order and the first one that can handles it. If a chain of catch blocks contains one that uses `catch_any` as the filtering tag, the rest of the catch blocks will never have the chance to catch an exception.

To make the exception handling embedded language complete we can give the `exception` data-constructor a new name, `throw_`. Using it, returning an exception from a template metafunction is similar to throwing an exception in runtime code.

In runtime C++ code functions that are not prepared to handle exceptions can be called from `try` blocks without any further syntactic elements. When using monads, non-monadic operations need to be *lifted* [21] into the monad. We have specified our exception handling monad in a way that every value in template metaprogramming is a monadic value to avoid lifting when using compile-time exceptions and make it more like exceptions in runtime C++ code.

6.2. IMPLEMENTATION OF EXCEPTION HANDLING

The `try_` template can be implemented in two phases. As the first phase a `do_try` template can be implemented as a special `do` block prepared for exception handling. This is a `do` block of the Exception monad, but provides the `catch_` cases, thus it can be used the following way:

```
template <class A, class B> struct min :
  do_try<
    set<t, less<A, B>>,
    if_<t, A, B>
  >
    ::template catch_<comparison_error_tag, e>
      ::template apply< /* handle comparison error */ >
    ::template catch_<catch_any, e>
      ::template apply< /* handle other errors */ >
{};
```

The above code returns the result of the `do` block when it is not an exception and checks the `catch_` cases otherwise. As the second phase the `try_` template can be implemented that takes an angly bracket expression as its argument and turns it into a `do_try` block.

A `try_` block has to turn an angly bracket expression into a monadic chain of function calls. In our `min` example, this angly bracket expression is `if_<less<A, B>, A, B>`. Since an angly bracket expression describes a type, it is either a non-template type or an instance of a template class. When it is a non-template type, we can not transform it further, thus it can be turned into a one-step `do_try` block:

```
template <class F> struct try_ : do_try<F> {};
```

When it is an instance of a template class, we assume that all of the template arguments are nullary metafunctions that need to be evaluated first. We evaluate them one by one using different steps of the `do_try` block, we store their results using `set` elements in the `do_try` block and we evaluate the top-level template class as the final step of the `do_try` block. In the implementation we make use of partial specialisation of template classes and template template arguments [35].

```
template <
  template <class, class> class F,
  class T1, class T2>
struct try_<F<T1, T2>> :
  do_try<
    set<t1, T1>,
    set<t2, T2>,
    F<t1, t2>> {};
```

The above code shows how to prepare `try_` for angly bracket expressions, where the top-level template takes two arguments. For every metafunction arity a different specialisation needs to be written. These specialisations can be generated using the Boost.Preprocessor library [39], we do not present the details of it here. We have shown how to implement the second phase, turning `try_` blocks into `do_try` blocks.

To implement the first phase as well, we present our solution for extending a `do` block for the Exception monad with catching exceptions. A `do_try` - `catch_` construct can be modeled using a finite state machine [9] with two states:

- `was_exception` The result of the do block was an exception and this exception has not been handled yet.

- `skip_further_catches` The result of the do block was not an exception or it was an exception but has already been handled.

The initial state of the machine depends on the result of the do block. Both states are end states, the machine can stop in any of them. A state is implemented by a template class. State transitions are applications of the `catch_` nullary metafunctions. Thus, state transitions are implemented by instantiating nested `catch_` template classes of the results of `catch_` metafunction classes. State transitions are implemented by inheritance: the `apply` template class inside `catch_` inherits from the next state. The implementation of `skip_further_catches` is the following:

```
template <class Result>
struct skip_further_catches {
  typedef Result type;
  template <class Tag, class Name> struct catch_ {
    template <class Body>
    struct apply : skip_further_catches {};
}; };
```

When there was no exception or after the exception has been handled, all further exception handling blocks should be skipped. At the end of the chain of `catch_` applications the user of the `do_try` block will need access to the result, thus `type` is a `typedef` of the result. The implementation of `was_exception`:

```
template<class Result> struct lazy_skip_further_catches:
  skip_further_catches<typename Result::type> {};
template <class Exception> struct was_exception {
  typedef Exception type;
  template <class ExceptionTag, class Name>
  struct catch_ {
    template <class Body>
    struct apply : boost::mpl::if_<
      boost::mpl::or_<
        is_same<ExceptionTag, typename boost::mpl::tag<
          typename get_data<Exception>::type>::type>,
        is_same<ExceptionTag, catch_any>>,
      lazy_skip_further_catches<typename
        let<Name, typename get_data<Exception>::type,
          Body>::type>,
      was_exception>::type {}; }; };
```

It checks if the `tag` of the exception and the expected `tag` are the same or the expected `tag` is `catch_any`. In both cases it evaluates the body of the catch block and transitions to the `skip_further_catches` state. Otherwise it remains in the `was_exception` state. It uses a helper metafunction, `lazy_skip_further_catches` to avoid evaluation of the exception handling code in those cases, when the `tag`s do not match. `do_try` can be implemented using these states:

```
template <class step1, ..., class stepn>
struct do_try :
  boost::mpl::if_<
    typename boost::is_same<
      exception_tag,
      typename boost::mpl::tag<
        typename do_<exception_monad>
          ::template apply<step1, ..., stepn>::type
      >::type
    >::type,
    was_exception<
      typename do_<exception_monad>
        ::template apply<step1, ..., stepn>::type
    >,
    skip_further_catches<
      typename do_<exception_monad>
        ::template apply<step1, ..., stepn>::type
    >
  >::type {};
```

It evaluates the body of the `do_try` block using a `do_` block and starts the state machine from one of its states depending on the result of the `do_` block. Support for multiple arguments can be implemented using the Boost.Preprocessor library, the details of which are not presented here. The full implementation can be found in [44].

6.3. MEASURING THE EXCEPTION HANDLING SOLUTION

We present the implementation of the `min` function with and without using the `try_` blocks we have presented. Here is the implementation of it without using `try_` blocks:

```
template <class A, class B> class min {
private:
  template <class LessAB> struct impl :
    boost::mpl::if_<typename LessAB::type, A, B> {};
public:
  typedef typename boost::mpl::eval_if<
    typename is_exception<less<A, B>>::type,
    less<A, B>, impl<less<A, B>>> type;
};
```

The above code implements the logic of calculating the minimum of two values and deals with error propagation at the same time, which makes it difficult to follow. The same can be implemented using a `try_` block:

```
template <class A, class B>
struct min : try_<boost::mpl::if_<less<A,B>, A,B>> {};
```

The above code implements the same metafunction as the previous one, except that it uses an Exception monad to deal with error propagation and the rest of the code can focus on the business logic – calculating the minimum of two values in this case.

We have measured the cost of instrumenting template metaprograms with exception propagation. We measured the compilation speed of template metaprogramming expressions embedded and not embedded in a try block. We were using the `plus` metafunction from Boost.MPL to construct the expressions. We used `plus<int_<1>, int_<N>>` as the expression. We calculated this expression a hundred times in one compilation by replacing `N` with a value between 1 to 100. We have run the measurement several times and we increased the complexity of the expression by adding further `plus` elements:

```
  plus<int_<1>, int_<N>>
  plus<int_<1>, plus<int_<1>, int_<N>>>
  // ...
```

We run the measurements on a Linux command line. The machine we did the measurements on had an 1.6 GHz Atom processor and 1 GB memory. We were using gcc 4.5.2 and we didn't use any optimization. Figure 1 shows the results. Complexity on the diagram means the number of `plus` elements in the expression.

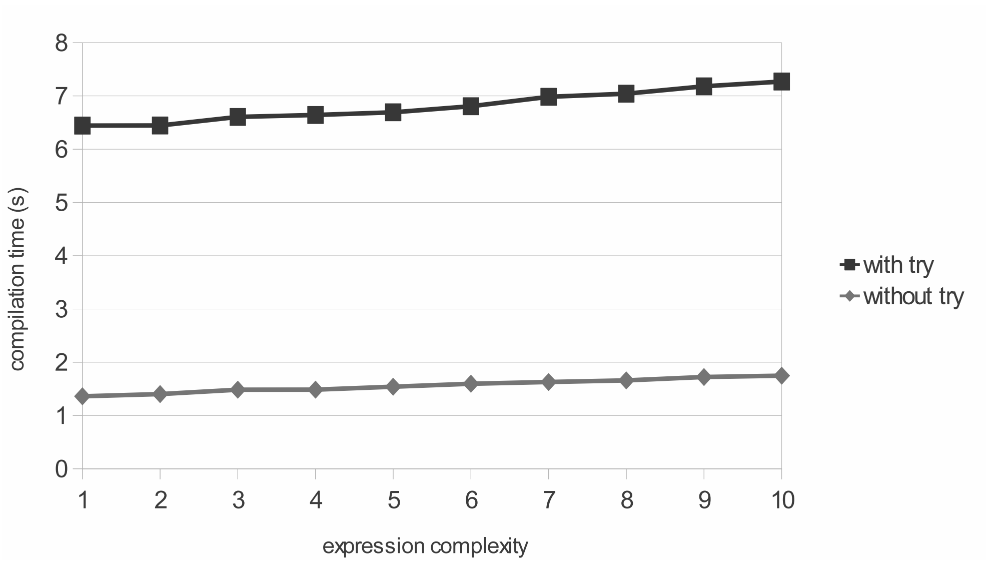

Figure 1: Comparison of compilation speed with and without exception handling

The results show that even though there is a significant difference between using and not using a try block, by increasing the complexity of the instrumented expression the compilation time doesn't grow faster by using try blocks.

Using this approach adding proper error handling to template metaprograms is easy and developers not familiar with monads can also understand it. An implementation of this approach is part of the Mpllibs [44] library collection.

6.4. PARSER MONAD

In Domain-specific Language Integration with Compile-time Parser Generator Library [23] we present an approach for implementing parsers that parse an embedded DSL script at C++ compilation time. Error messages, classes or runtime code can be generated as a result of parsing. The Domain-specific Language Integration paper uses parser combinators [3] to build parsers. The approach presented there uses a monadic approach for handling parsing errors, however it doesn't use a framework supporting monads in template metaprogramming. We show how a monadic framework, such as the one presented in this paper can simplify parser construction.

The paper introduces the concept of a parser, which is a function taking a compile-time string as input and returning either a special value, `nothing`, or a pair of a result and the remaining string. We do not present parser combinators here in detail, we assume that the reader is already familiar with it. Here is an example grammar:

```
EXP ::= NUMBER '+' NUMBER
```

To build a parser for it one has to build a parser for `NUMBER`, one for `'+'` and combine them in a way that they are applied on the input string in order. Each sub-parser takes the yet unparsed suffix of the input string and returns some result (eg. the parser for `NUMBER` returns the value of the parsed number) and the unparsed suffix of the input.

A monadic framework helps combining functions into more complex functions in a readable way. The fact that parser combinators are about combining parsers – which are functions – into more complex parsers suggests that monads could make parser construction easier.

Parsers operate on the input string: a parser consumes a part of the input, other parsers following it work on this updated input. For example to parse an `EXP` we need to give the initial input string to the `NUMBER` parser to parse the number on the left. It returns the parsed value and the unparsed suffix of the input. This unparsed suffix needs to be passed to the parser parsing `'+'` and so on. Without a monadic framework it is the job of the developers combining parsers to pass this input string around. However, the State monad can deal with this if used in a way that the input string is the state.

Parsers need to handle errors as well: when a parser fails in a chain of parsers, the chain needs to be stopped and the error needs to be propagated. This error propagation logic can be handled by the Maybe or the Either monad.

We have shown that there are two different monads making parser construction easier. In this section we present a monad that provides these two capabilities together. Using monad transformers [21] this monad could be built from the State and the Maybe or the Either monad. Monad transformers are out of scope for this paper.

A new monad, the Parser monad can be implemented using the approach we present in this paper for the parsers:

- Monadic values are parsers, that is, functions.
- `return_` constructs the `return_` parser described in the paper. It consumes no input and returns the argument of `return_` as the parsing result.
- `bind` constructs a parser that parses the input using `bind`'s first argument, passes the result to the second argument of `bind` to get a new parser. It parses the remaining string using this new parser and returns the result of it. When the first parser fails, the error is propagated.

Using this monad error propagation can be simplified in the implementation of parsers and parser combinators. The Domain-specific Language Integration paper presents the `accept_when` parser combinator that takes a parser, `M`, and a predicate `P`. The combinator builds a new parser that parses the input using `M` and accepts the result if and only if the `P` predicate returns `true` for the result. When `M` fails or `P` returns `false`, the parser rejects the input. It is implemented the following way:

```
template <class M, class P> struct accept_when {
  struct type {
    template <class Cs> struct apply : lazy_eval_if<
      equal_to<typename apply<M, Cs>::type, Nothing>,
      nothing, lazy_eval_if<
        apply<P, just_value<apply<M, Cs>>>,
        apply<M, Cs>, nothing
      >> {};
  };
};
```

The above parser can be re-written using the Parser monad we have implemented based on the technique presented in this paper.

```
struct r;

template <class M, class P> struct accept_when :
  do_<parser>::apply<
    set<r, M>,
    lazy_if<apply<P, r>, do_return<r>, fail>
  > {};
```

Note that `fail` is a parser that fails to parse any input. This solution implements the propagation of the failure using the Parser monad. Using the monad, the implementation of `accept_when` became shorter due to the removal of error propagation logic and state passing.

We have added this monad to the Mpllibs library collection [44]. Note that our implementation is slightly more complicated to be able to provide more detailed information about the error. However, the additional complexity can be hidden by the Parser monad, thus code constructing parsers using this monad remains as simple as presented in this paper.

7. RELATED WORK

The connection between Haskell, C++ template metaprogramming and using monads has been discussed several times.

- Bartosz Milewski talked about the connection between Haskell and C++ template metaprogramming in his blog [17]. He presented the similarities of the two languages. He explained monads [19] and that they could be used in C++ [18]. He discussed the relationship between monads and exception handling. He did not use monads to implement exception handling in compile-time C++ code. His solution used functor objects to implement the functions `bind` connects and he presented the Reader monad as well. His approach focuses on using monads at runtime, while our approach focuses on using them at compile-time.

- FC++ [16] is a library providing functional programming tools for C++ supporting monads. Each monad is represented by a C++ class, which specifies the required operations, bind and unit (our return). It provides a do notation implementation, as well, based on C++ operator overloading. The library provides tools for developing programs in the functional style that are executed at runtime, while our solution focuses on monads for calculations at compile time.

- Joaquín M López Muñoz discussed using monads in template metaprograms in his blog [14] and presented a simple implementation. His solution has many similarities with the solution presented in this paper. He created metafunctions for `bind` and `return`. His solution is based on overloading these metafunctions using pattern matching and does not take `tags` into account.

While our `bind` implementation takes the tag of the monad as argument, his `mbind` operation determines the monad from its arguments, which is closer to the way monads in Haskell work. However, due to the lack of using tags, his solution cannot deal with different template classes implementing the same data-structure.

To compare Muñoz's approach to ours, we compare the implementation of the List monad using the two approaches. We have already presented the implementation of the List monad using our approach in Section 4.3. Given the fact that Muñoz's approach is based on pattern matching, to implement the List monad we need a list implementation where we can do pattern matching. We can use the following:

```
struct empty;
template <class Head, class Tail> struct cons {};
```

`empty` represents the empty list, `cons<Head, Tail>` represents a list with `Head` as its first element and `Tail` as the remaining list. Based on this, we can implement the List monad's operations:

```
template <class T> struct mreturn<empty, T>
{ typedef cons<T, empty> type; };

template <class A, class B> struct list_append;
template <class B> struct list_append<empty, B>
{ typedef B type; };
template <class H, class T, class B>
struct list_append<cons<H, T>, B> {
  typedef
    cons<H, typename list_append<T, B>::type> type;
};

template <class F>
struct mbind<empty, F> { typedef empty type; };

template <class H, class T, class F>
struct mbind<cons<H, T>, F> : list_append<
  typename boost::mpl::apply<F, H>::type,
  typename mbind<T, F>::type> {};
```

The above code implements the List monad for the list implementation given above. Since Muñoz's approach is based on pattern matching, the List monad implementation one can provide is limited to using this (or some other) list implementation. Using the monad implementation approach presented in this paper, we could implement the List monad in a way that is not tied to any actual list implementation. It is based on the polymorphic list operations of Boost.MPL and tag dispatching, thus it works with any list implementation.

We have compared how monads can be constructed using the two approaches. We haven't compared how the monads can be used and how well the two approaches can simplify the complex syntax of template metaprogramming. Given that we have provided an implementation of Haskell's do syntax while there is no implementation of it for Muñoz's approach, there is no reasonable way to compare the two approaches.

- Norman Ramsey [24] used a monadic approach to avoid meaningless error messages caused by earlier errors in a compilation process. His approach focuses on the improvement of error reporting in compilers. He implemented his approach in ML and used monads for implementing error propagation.

- Mike Spivey [32] presents how built-in exception handling can be replaced by a monadic approach in functional languages. As Norman Ramsey mentions it [24], this technique can be used in other languages, including C++. We have presented a similar approach in this paper, however Spivey's approach is based on the Maybe monad while our approach is based on the Either monad and can return error details as well, not just the fact that an exception was thrown.

- Stuart Golodetz talks about the connection between functional programming and C++ template metaprogramming [7]. He presents the connection between Haskell and C++ template metaprograms by converting Haskell lists and functions operating on them to C++ template metaprograms. In the second part [8] he presents the implementation of balanced trees in C++ template metaprogramming. He doesn't talk about monads.

- *phaskell* [43] and *MetaFun* [42] are translators converting Haskell-like languages to C++ template metaprograms. They use a simple sub-language of Haskell, which is enough for implementing simple functions and data-structures, however they do not support typeclasses, thus they can not be used to implement monads in C++ template metaprogramming.

- Jean-Philippe Bernardy et al. present the connection between Haskell type classes and C++ concepts [4]. Unfortunately C++ concepts are not part of the new C++ standard and cannot be used to implement type classes in standard C++.
- Ádám Sipos presents Meta<Fun> [31], which is a translator from a Clean-like pure functional language to C++ template metaprograms. The tool consists of an execution engine, a C++ template metaprogram library, and an external translator. The paper does not mention monads.
- Graham Hutton and Erik Meijer introduce parser combinators and monads by giving a tutorial on how to construct a monadic parser combinator library. [11, 12] The paper discusses the performance of parsers built with parser combinators. Techniques improving the performance of the generated parses are also presented. The paper presents different monads (Maybe, List, State monads) and how they can be combined in the resulting parser combinator library.
- Dan Popa presents a step-by-step process [22] on how to construct an interpreter in Haskell using monads. He constructs the interpreter in two stages. In the first stage he builds the back-end, which executes the interpreted program. Monads are used for maintaining the state and producing the output of the interpreter. In the second stage he builds the front-end, which parses the source code of the program to interpret using a monadic parser combinator library, Parselib.
- Tim Sheard at al. present a method for domain specific language construction. [25] They cover the construction of such languages from the domain analysis phase to the implementation. Monads play a significant role in the process. The authors suggest designing with monads to deal with side-effects and state. The other reason they suggest using monads is to make the resulting code cleaner and more readable by hiding a large amount of plumbing with the help of the monads. They present their solution in MetaML.

8. CONCLUSION

Recognizing the similarities between Haskell and C++ template metaprogramming as pure functional languages, we have adopted Haskell monads in metaprograms and implemented a framework supporting them as a library. We have demonstrated how the library works by implementing several monads from Haskell to C++ template metaprogramming.

We have seen a major difference between the C++ template metaprogramming and the Haskell implementation of typeclasses that affects monads as well: while in Haskell the compiler verifies that a type meets the requirements of a class when it is made an instance of the class, in C++ we do not get this guarantee. In many situations the lack of such verifications is a disadvantage, since errors related to that are delayed until the first time someone tries to use the missing part of the typeclass. However, we have seen that in some cases the lack of such verifications can be useful. We could implement exception handling in C++ template metaprogramming in a way that doesn't fit into the Haskell type system – we have made every value a monadic value.

We have shown two real world use cases. We have implemented a generic error propagation solution for C++ template metaprograms using monads and an embedded language for compile-time exception handling on top of that. As another real world use case we have shown how the implementation of a compile-time parser generator library can be simplified by using monads.

References

[1] D. Abrahams, A. Gurtovoy, C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, Boston, 2004.

[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

[3] Lennart Anderson, Parsing with Haskell, 2001, `http://www.cs.lth.se/eda120/assignment4/parser.pdf`

[4] Jean-Philippe Bernardy and Patrik Jansson and Macin Zalewski and Sibylle Schupp, Generic programming with c++ concepts and haskell

type classes: A comparison, In J. Funct. Program., Cambridge University Press, New York, NY, USA, volume 20, issue 3-4, pp. 271–302. `http://dx.doi.org/10.1017/S095679681000016X`.

[5] K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.

[6] Y. Gil, K. Lenz, Simple and Safe SQL queries with C++ templates, In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.

[7] Stuart Golodetz, Functional Programming Using C++ Templates (Part 1), Association of C and C++ Users, 2007, `http://www.accu.org/var/uploads/journals/overload81.pdf`

[8] Stuart Golodetz, Functional Programming Using C++ Templates (Part 2), Association of C and C++ Users, 2007, `http://www.accu.org/var/uploads/journals/Overload82.pdf"`

[9] J. E. Hopcroft, R. Motwani, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1969.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns. Addison-Wesley. 1995. ISBN 0-201-63361-2.

[11] Graham Hutton and Erik Meijer, Monadic Parser Combinators, Technical Report, Department of Computer Science, University of Nottingham, NOTTCS-TR-96-4, 1996

[12] Graham Hutton and Erik Meijer, Monadic Parsing in Haskell, Journal of Functional Programming, Cambridge University Press, 1998. Volume 8, pp. 437–444.

[13] Björn Karlsson, Beyond the C++ Standard Library: An Introduction to Boost, Addison Wesley Professional, 2005.

[14] Joaquín M López Muñoz, Monads in C++ template metaprogramming, `http://bannalia.blogspot.com/2008/06/monads-in-c-template -metaprogramming.html`

[15] B. McNamara, Y. Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000

[16] Brian McNamara, Yannis Smaragdakis: Functional programming in C++. Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.

[17] Bartosz Milewski, Haskell and C++ template metaprogramming `http://bartoszmilewski.wordpress.com/2009/10/26/haskellc-video-and-slides`

[18] Bartosz Milewski, Monads for the Curious Programmer `http://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1`

[19] Bartosz Milewski, Monads in C++, `http://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1/`

[20] Nathan Myers, A new and useful template technique: "traits", C++ gems, SIGS Publications, Inc., New York, NY, USA, 1996, ISBN 1-884842-37-2, pp. 451–457.

[21] B. O'Sullivan, J. Goerzen, D. Stewart, Real World Haskell, O'Reilly, 2008. ISBN: 978-0-596-51498-3

[22] Dan Popa, How to Build a Monadic Interpreter in One Day, In Stud. Cercet. Stiint., Ser.Mat., Supplement Proceedings of CNMI 2007, Cambridge University Press, Volume 17, pp. 173–192.

[23] Zoltán Porkoláb and Ábel Sinkovics, Domain-specific language integration with compile-time parser generator library, In Eelco Visser and Jaakko Järvi, editors, Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, ACM, October 10-13, 2010,

[24] Norman Ramsey, Eliminating Spurious Error Messages Using Exceptions, Polymorphism, and Higher-Order Functions, In Computer Journal, Volume 42, 1999.

[25] Sheard, Tim and Benaissa, Zine-el-abidine and Pasalic, Emir, DSL implementation using staging and monads, In SIGPLAN Not., volume 35, issue 1, New York, NY, USA, ACM, December 1999, pp. 81–94.

[26] J. Siek and A. Lumsdaine: Concept checking: Binding parametric polymorphism in C++, In First Workshop on C++ Template Metaprogramming, October 2000

[27] Ábel Sinkovics, Functional Extensions to the Boost Metaprogram Library, In Electr. Notes Theor. Comput. Sci., 264(5), pp. 85–101, 2010.

[28] Ábel Sinkovics, Nested Lambda Expressions with Let Expressions in C++ Template Metaprograms, In Electr. Notes Theor. Comput. Sci., 279(3), PP. 27-40, 2011.

[29] Ábel Sinkovics, Functional extensions to the Boost Metaprogram Library, In Porkolab, Pataki (Eds) Proceedings of the 2nd Workshop of Generative Technologies, WGT'10, Paphos, Cyprus. pp.56–66 (2010), ISBN: 978-963-284-140-3

[30] Sinkovics, Ábel, Nested Lambda Expressions with Let Expressions in C++ Template Metaprograms in (Eds) Zoltán Porkoláb, Norbert Pataki: Proceedings of 3rd Workshop on Generative Technologies, WGT'11, Saarbrücken, Germany, 2011, pp. 63–76, ISBN: 978-963-284-188-5.

[31] Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms, in Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), pp. 489–502.

[32] Spivey, M., A functional theory of exceptions, In Sci. Comput. Program., Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, 14(1), May, 1990, pp. 25–42

[33] Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)

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

[35] D. Vandevoorde, N. M. Josuttis, C++ Templates: The Complete Guide, Addison-Wesley, 2003.

[36] T. Veldhuizen, Expression Templates, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[37] T. Veldhuizen, D. Gannon, Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23

[38] T. Veldhuizen, Using C++ Template Metaprograms, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[39] Vesa Karvonen and Paul Mensonides, The Boost preprocessor library. `http://www.boost.org/doc/libs/1_47_0/libs/preprocessor/doc`

[40] Aleksey Gurtovoy and David Abrahams, The boost metaprogramming library.
`http://www.boost.org/doc/libs/1_47_0/libs/mpl/doc/index.html`

[41] Eric Niebler, The boost xpressive regular library.
`http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html`

[42] Haskell to C++ Template metaprogramming translator,
`http://gergo.erdi.hu/projects/metafun/`

[43] Haskell to C++ Template metaprogramming translator,
`http://code.google.com/p/phaskell/w/list`

[44] Ábel Sinkovics, The source code of mpllibs
`http://github.com/sabel83/mpllibs`