Implementing Monads for C++ Template Metaprograms*

Technical report TR-01/2011

Ábel Sinkovics and Zoltán Porkoláb
Eötvös Loránd University,
Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C
Budapest, Hungary {abel, gsd}@elte.hu

September 2, 2011

*The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

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.

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 [37]. These compile-time programs are called *C++ template metaprograms* and form a Turing-complete sub language of C++ [8].

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, 28]; *active libraries* [38], acting dynamically at compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [9] for type-safe SQL interface, Boost.Xpressive [22] for regular expressions or Metaparse [25] for parsing at compile-time.

Initially C++ template metaprograms were constracted 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 [8]. 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 can not 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 functions.

One can look at template metaprogramming as a pure functional language [8]. Based on this similarity, a number of useful techniques for functional languages, such as Haskell [23], can be adopted in C++ template metaprogramming.

In Haskell, a monad [23] 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* [23] 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.

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. 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 [25]. The library was designed in a monadic way, however it didn't use any framework supporting monads.

All solutions presented in this paper are implemented and available as an open source library [3].

The rest of the paper is organized as follows. In Section 2 we implement algebraic data types and type classes of Haskell in C++ template metaprogramming. In Section 3 we explain our implementation of Haskell monads in C++ template metaprogramming. A number of well-known Haskell monads and their C++ template metaprogram implementations are discussed in Section 4. The two real world examples are presented in Section 5. Related research results are discussed in Section 6. Our paper concludes in Section 7.

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 [25] 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; // It is 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 [23] for implementing function overloading. There is a known similarity between Haskell typeclasses and C++ concepts [7], however, concepts have been removed from the new standard. We present a solution for implementing typeclasses in the old (and the new) 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 Comparable a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
```

This example defines a typeclass called `Comparable`. The argument of this typeclass 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 Comparable 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 Comparable 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* [21]. 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 can not be explicitly encoded. The following example shows the `Comparable` typeclass implemented in template metaprogramming:

```
template <class A>
struct 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 [12] uses *tag*s to implement template metafunction overloading [4]. 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` [4]. The following example shows how to make the boxed integers of Boost.MPL instances of our example typeclass.

```
template <>
struct 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 `comparable` template class for `integral_c_tag` and implements the 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 `comparable` typeclass in both languages:

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

// Template metaprogramming
template <class X>
struct self_equal : boost::mpl::apply<
  typename 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 comparable_defaults
{
  struct not_equal
  {
    template <class A, class B>
    struct apply : boost::mpl::not_<
        typename boost::mpl::apply<
          typename comparable<A>::equal, A, B
        >::type
    > {};
  };
};

template <>
struct comparable<integral_c_tag> :
  comparable_defaults<integral_c_tag>
{
  struct equal
  {
    // Same as before...
  };
  // not_equal is inherited from comparable_defaults
};
```

The default implementation of `not_equal` uses the `equal` method of struct `comparable`. 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 providing 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't 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 [23]

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't 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 [29], 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't 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:

```
using namespace boost;
template <class T>
struct monadic_error {};

template <class Tag>
struct monad_defaults
{
  struct bind_
  {
    template <class A, class B>
    struct apply : mpl::apply_wrap2<
      typename monad<Tag>::bind,
      A,
      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 don't 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 [23] that are documented but can not 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 `mpl::apply<F,X>`.
- *right identity*: `bind<Tag,M,monad<Tag>::return_>` is equivalent to `M`.
- *associativity*: `bind<Tag,M,lambda<x, bind<Tag, mpl::apply<F,x>, G> > >` is equivalent to `bind<Tag, bind<Tag, M, F>, G>`. `lambda` is our lambda expression implementation [31].

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 [3].

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't be implemented as a type in template metaprogramming, only as a set of individual data-constructors. Template metaprogramming can't 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 don't present it here. We need a new tag representing the `Maybe` monad:

```
struct maybe {};
```

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

```
using namespace boost;
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 : mpl::apply<
      F,
      typename get_data<A>::type
    > {};

    template <class A, class F>
    struct apply : mpl::if_<
      is_nothing<A>,
      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't 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 don't present it here. We need to create a new tag, `either`, for the `Either` monad. We can make `either` an instance of `monad`:

```
using namespace boost;
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 : mpl::apply<
      F,
      typename get_data<A>::type
    > {};

    template <class A, class F>
    struct apply : mpl::if_<
      is_left<A>,
      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 turning lists to 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`. The implementation of these operations is trivial, we don't present it here. It can be found in Mpllibs [3]. The list monad can be used to implement ambiguity in pure code [13, 14].

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`:

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

```
          typename 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 return `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.

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

```
      };
    };
  };

  struct bind
  {
    template <class A, class F>
    struct impl
    {
      template <class S> class apply : mpl::apply<
        typename mpl::apply<
          F,
          typename apply_first::first
        >::type,
        typename 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 in C++ template metaprograms.

4.6 Writer

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 [23] 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 [3]. The monadic values of the Writer monad are pairs: a value and a state. The states are expected to form a monoid [23], 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:

```
using namespace boost;

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

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

    typedef mpl::pair<
      typename FA_first::first,
      typename mpl::apply<
        typename monoid<Monoid>::append,
        typename A::second, typename FA_first::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 Use Cases for 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.

5.1 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 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 a 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 [30]. `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, that 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 [23]

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 between 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 for 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 types of 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't 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 exception 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* [23] 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.

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

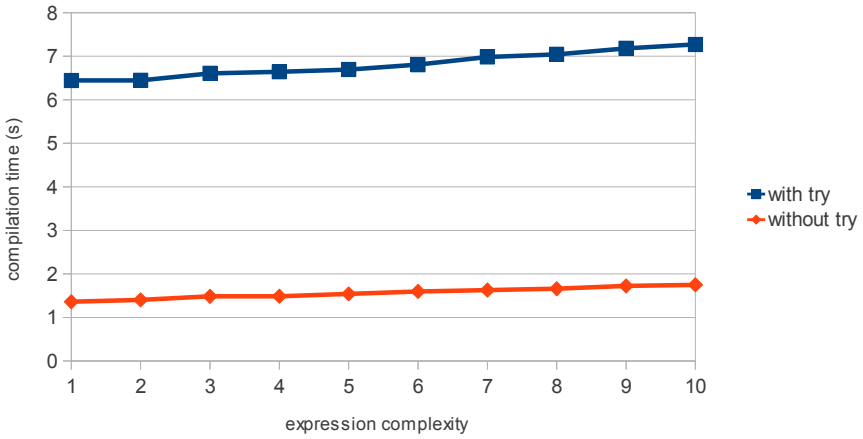

Figure 1: Comparison of compilation speed with and without exception handling

the results. Complexity on the diagram means the number of `plus` elements in the expression.

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 [3] library collection.

5.2 Parser Monad

In Domain-specific Language Integration with Compile-time Parser Generator Library [25] 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 paper uses parser combinators [6] 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.

The paper introduces the concept of a parser, that 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. A new monad, the Parser monad can be implemented using the approach presented 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 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 : bind<
  parser,
  M,
  lambda<
    r,
    lazy_if<apply<P,r>, return_<parser,r>, fail<parser> >
  >
>::type {};
```

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.

We have added this monad to the Mpllibs library collection [3]. Note that our implementation is slightly more complicated to be able to provide more detailed information about the error.

6 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 [18] and that they could be used in C++ [19]. 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 [20] 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 `tag`s 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.

- Norman Ramsey [26] 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 [26], 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 Golodotz talks about the connection between functional programming and C++ template metaprogramming [10]. 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 [11] he presents the implementation of balanced trees in C++ template metaprogramming. He doesn't talk about monads.

- *phaskell* [1] and *MetaFun* [2] are translators converting Haskell-like languages to C++ template metaprograms. They use a simple sublanguage of Haskell, which is enough for implementing simple functions and data-structures, however they don't support typeclasses, thus they can't be used to implement monads in C++ template metaprogramming.

- Jean-Philippe Bernardy et al. present the connection between Haskell type classes and C++ concepts [7]. Unfortunately C++ concepts are not part of the new C++ standard and can not be used to implement type classes in C++.

- Ádám Sipos presents Meta¡Fun¿ [5], 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. [13, 14] 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 [24] on how to construct an interpreter 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. [27] 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.

7 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 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] Haskell to c++ template metaprogramming translator, 2010.

[2] Haskell to c++ template metaprogramming translator, 2010.

[3] Ábel Sinkovics. The source code of mpllibs, 2010.

[4] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[5] Ádám Sipos, Zoltán Porkoláb, and Viktória Zsók. Meta¡fun¿ - towards a functional-style interface for c++ template metaprograms. *Studia Universitatis Babes-Bolyai Informatica*, LIII(2008/2):55–66, 2008.

[6] Lennart Andersson. Parsing with haskell, 2001.

[7] Jean-Philippe Bernardy, Patrik Jansson, Macin Zalewski, and Sibylle Schupp. Generic programming with c++ concepts and haskell type classes: A comparison. *J. Funct. Program.*, 20:271–302.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] Joseph (Yossi) Gil and Keren Lenz. Simple and safe sql queries with c++ templates. *Sci. Comput. Program.*, 75:573–595, July 2010.

[10] Stuart Golodetz. Functional programming using c++ templates (part 1). *Overload*, (81), October 2007.

[11] Stuart Golodetz. Functional programming using c++ templates (part 2). *Overload*, (82), December 2007.

[12] Aleksey Gurtovoy and David Abrahams. Boost.mpl, 2004.

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

[14] Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[15] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *C++ Template Programming Workshop*, October 2000.

[16] Brian McNamara and Yannis Smaragdakis. Functional programming in c++ using the fc++ library. *SIGPLAN Notices*, 36(4):25–30, 2001.

[17] Bartosz Milewski. What does haskell have to do with c++?, 2009.

[18] Bartosz Milewski. Monads in c++, 2011.

[19] Bartosz Milewski. Monads in c++, 2011.

[20] Joaquín M López Muñoz. Monads in c++ template metaprogramming, 2008.

[21] Nathan Myers. *A new and useful template technique: "traits"*, pages 451–457. SIGS Publications, Inc., New York, NY, USA, 1996.

[22] Eric Niebler. Boost.xpressive, 2007.

[23] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[24] Dan Popa. How to build a monadic interpreter in one day. *Stud. Cercet. Stiint., Ser.Mat., Supplement Proceedings of CNMI 2007*, 17:173–192, 2007.

[25] Zoltán Porkoláb and Ábel Sinkovics. Domain-specific language integration with compile-time parser generator library. In Eelco Visser and Jaakko Järvi, editors, *GPCE*, pages 137–146. ACM, 2010.

[26] Norman Ramsey. Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions. *Computer Journal*, 42, 1999.

[27] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. Dsl implementation using staging and monads. *SIGPLAN Not.*, 35:81–94, December 1999.

[28] Jeremy G. Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[29] Ábel Sinkovics. Functional extensions to the boost metaprogram library. *Electr. Notes Theor. Comput. Sci.*, 264(5):85–101, 2010.

[30] Ábel Sinkovics. Functional extensions to the boost metaprogram library. In Zoltán Porkoláb and Norbert Pataki, editors, *WGT'10*, volume II of *WGT Proceedings*, pages 56–66. Zolix, 2010.

[31] Ábel Sinkovics. Nested lamda expressions with let expressions in c++ template metaprorgams. In Zoltán Porkoláb and Norbert Pataki, editors, *WGT'11*, volume III of *WGT Proceedings*, pages 63–76. Zolix, 2011.

[32] M. Spivey. A functional theory of exceptions. *Sci. Comput. Program.*, 14:25–42, May 1990.

[33] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[34] Erwin Unruh. Prime number computation, 1994.

[35] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 1 edition, November 2002.

[36] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[37] Todd Veldhuizen. *Using C++ template metaprograms*, pages 459–473. SIGS Publications, Inc., New York, NY, USA, 1996.

[38] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing OO'98*. SIAM Press, 1998.