Meta-programming High-Order functions

Abstract

This paper presents a proposal for some high-order template meta-programming functions based on some common patterns used in libraries as Meta and Boost.MPL.

Some of these utilities are used in the interface of P0338R2 and P0196R3 and other have been used in their respective implementations.

Table of Contents

1. Introduction
2. Motivation and scope
3. Proposal
4. Design rationale
5. Proposed wording
6. Implementability
7. Open points
8. Acknowledgements
9. References

History

Revision 1

- Fix typos
- Replace Meta-Callable by Meta-Invocable.
- Define *TypeConstructible* in function of `value_type_t`
- Update references.

**Introduction**

This paper presents a proposal for some high-order template meta-programming functions *Meta-Invocables* based on some common and well know patterns used in libraries as *Meta* and *Boost.MPL*.

Some of these utilities are used in the interface of [P0338R2](https://example.com) and [P0196R3](https://example.com) and other have been used in their respective implementations. In particular the following is used:

- `is_invocable<TC(Xs...)>
- `invoke<TC, Xs...>
- `type_constructor
- `is_type_constructor
- `quote<Tmpl>
- `quote_trait<Tmpl>
- `bind_back<Fn, Xs...>
- `rebind<T, Xs...>

Others are not used but added for completion, as

- `identity
- `always
- `bind_front

The following traits are really optional

- `eval
- `id

**Motivation and scope**

C++ has already class templates and template alias that can be seen as meta-programming function that build other types by instantiation of the template.

The C++ standard library has also type traits that add an additional level of indirection via the nested type alias `type`.

As any high-order function library we should be able to pass meta-programming function as parameters and return meta-programming functions. While the first is possible with class templates, we are unable to return them, we need an artifice, nest a class template `invoke` as the result of the returned class.
TinyMeta contains a good description of why high order function is as useful in meta-programming as it is in functional programs, at the end meta-programming is a functional language. Boost.MPL calls these high-order meta-programming functions Metafunction Classes. Meta call them Callable. The C++ standard has recently moved to use Invokabel in function of std::invoke, so we will use here Meta-Invocable.

One of the uses of Meta-Invocables as type constructors as any Meta-Invocables return in some way a type. We call them also type-constructors.

Boost.Hana takes a different direction. Instead of using meta-programming techniques, it uses usual C++14 constexpr functions and use a trick type_c<T> to pass types to these functions. It needs also the use of decltype to get the resulting type and then unwrap it, decltype(t)::type. The authors would like to see a concrete proposal using this alternative direction but prefer the Hana's author to do it.

P0196R3 and P0338R2 depends on this proposal.

Proposal

Type-Traits helpers

```
meta::id
```

Results always in its parameter.

Meta provides the same.

Boost.MPL calls this mpl::identity.

Boost.Hana has an hana::id constexpr function.

Example

```
template <class T>
struct value_type<optional<T>> : meta::id<T> { };
```

```
meta::eval
```

Template alias to shortcut the idiom typename T::type.

Meta used to name it meta::eval but name it now meta::t_.

Example
template <class M, class ...Us>
using rebind_t = eval<rebind<M, Us...>>;

**Meta-Invocables types**

**Requirements**

A *Meta-Invocable* is a class that has a nested template alias `invoke`.

*Meta* provides the same.

*Boost.MPL* It defines *MetaFunctionsClass* as something similar and requiring a nested `apply` type trait instead of nested `invoke` template alias.

*Boost.Hana* It defines *MetaFunctions* as something similar but adapted to the run-time function and instead of requiring `invoke` it requires a nested `apply` *Boost.Hana-Metafunction*

For example

```cpp
struct identity
{
    template <class T>
    using invoke = T;
};
```

`meta::invoke`

As applying the class template `invoke` is not user friendly `TC::template invoke<Xs...>` it is preferable to have a template alias that do that `invoke<TC, Xs...>`

*Meta* provides the same.

*Boost.MPL* calls this `mpl::apply`.

*Boost.Hana* doesn't needs it as it uses normal function call syntax.

Example

```cpp
static_assert(is_same<meta::invoke<int>, int>::value, "meta::invoke error");
```

**Basic operations**
**meta::identity**

Results always its `invoke` parameter.

**Meta** not supported to the author knowledge.

**Boost.MPL** calls this `mpl::identity`.

**Boost.Hana** calls the equivalent function `hana::id`.

Example

```cpp
invoke<conditional_t<is_integral<T>::value,
    meta::identity,
    meta::always<void>>,
    int, string>;
```

**meta::always**

Results always its `template` parameter. Is the constant *Meta-Invocable*.

**Meta** calls it `meta::id<T>`.

**Boost.MPL** calls sthis `mpl::always`.

**Boost.Hana** calls the equivalent function `hana::always`.

Example

```cpp
invoke<conditional_t<is_integral<T>::value,
    meta::identity,
    meta::always<void>>,
    int, string>;
```

**meta::compose**

Composes several *Meta-Invocables*.

**Meta** provides the same.

**Boost.MPL** not supported to the author knowledge.

**Boost.Hana** calls the equivalent constexpr run-time function `hana::compose`.

Example
The following functions bind some parameters for later invocation.

- `meta::bind_front`
- `meta::bind_back`

`Meta` provides the same.

`Boost.MPL` has lambdas and so it can implement `mpl::bind`. `Boost.Hana` provides only partial application via the constexpr run-time function `hana::partial`.

Example

```
invoke<bind_back<quote<expected>>, error_code>, int>>
```

**Other Meta-Invocable factories**

Other helper meta-functions are useful to transform a class template or a type trait on an `Meta-Invocables`.

- `meta::quote`
- `meta::quote_trait`

Example

```
invoke<compose<quote_trait<add_pointer>, quote<optional>>, int>>
```

`Meta` provides the same.

`Boost.MPL` has no `compose` function.

`Boost.Hana` provides it with `hana::template_` and `hana::metafunction`.

`Boost.Hana` provides also `hana::metafunction_class` that transforms a `Boost.MPL` MetafunctionClass into a Hana Metafunction.

**Traits**

```
meta::is_invocable<Fn(Args...), R>
```
Checks if the result of invoking the class \( T \) with the arguments \( \text{Args...} \) is convertible to \( R \).

This trait follows the syntax and semantics of `std::is_invocable`.

`Meta` not supported to the author knowledge.

`Boost.MPL` not supported to the author knowledge.

`Boost.Hana` not supported to the author knowledge.

Example

```cpp
static_assert<is_invocable<identity(T)>>::value;
```

`meta::is_type_constructor`

Sometime we don't have yet the arguments to invoke the `Meta-Invocable`, but we want to check that at least the class has a nested class template `invoke`

Check if the class has a nested class template `invoke`.

`Meta` provides a similar trait `meta::is_invocable`.

We have called it `is_type_constructor` as a `Meta-Invocable` is use to construct types.

Example

```cpp
static_assert<is_type_constructor<identity>::value>;
```

**TypeConstructible types**

Given a type we want to be able to get a type constructor that could be use to construct the same type using the `meta::invoke`.

- `meta::type_constructor<T>`: an `Meta-Invocables` that can be used to construct the type \( T \)

`Boost.MPL` has something similar to `type_constructor` trait `mpl::unpack_args`.

We say that a type \( T \) is `TypeConstructible` type if the `meta::type_constructor_t<T>` and `meta::value_type_t<T>` are well defined and the following conditions are satisfied

```cpp
invoke<type_constructor<T>, meta::value_type_t<T>> is T
```
Example

```cpp
class optional_t { // ... }
```

### TypeConstructible Product types

Given that a *Product type* P0327R2 gives access to the element types and its size via

- `product_type::size<T>`
- `product_type::element<T, I>`: the *I*th arg type that can be used to construct the type `T`

We say that a type `T` is *TypeConstructible Product type* if the `meta::type_constructor_t<T>` and `product_type::element_t<T, I>` are well defined and the following conditions are satisfied when `N` is `product_type::size<T>`

```cpp
invoke<type_constructor<T>, product_type::element_t<T, 1>, ..., product_type::element_t<T, N>>;
```

### Rebindable types

When we have a type, it is often useful to rebinding the arg types to construct a similar type with the same type constructor.

The standard provides already something similar for *Allocator* via the `A::template rebind<T>::other` expression.

Most of the template classes can be rebound, as e.g. `optional`. Let call those types *Rebindable types*.

We want the following to be satisfied

```cpp
rebind_t<invoke<TC, Xs...>, Ys...> is the same as invoke<TC, Ys...>
rebind_t<Tmpl<Xs...>, Ys...> is Tmpl<Ys...>
invoke<type_constructor_t<T>, Xs...> is the same as rebind_t<T, Xs...>
rebind_t<rebind_t<T, Ys...>, Ys...> is the same as rebind_t<T, Ys...>
```

Any *TypeConstructible* type can be rebound using
But this is not friendly.

This paper propose to define rebind in function of a nested template alias `rebind` and defines a partial specialization for any class template having types as template parameters.

Alternatively we could define `rebind` as an alias of the previous expression.

Example

```cpp
static_assert(is_same<rebind_t<optional<int>, char>, optional<char>>::value);
```

### What is not proposed yet?

Other functional facilities will also be welcome, but this paper prefer to start with something concrete that is needed by other proposals.

**Lambdas**

It is also useful to be able to describe high-order meta-functions using meta-lambda expressions, but this paper let this facilities for another proposal.

**Type list**

Sometimes the type arguments are stored on a type list and so we need to unpack the list them before invoking.

- `meta::apply`: applying a *Meta-Invocables* to the elements of a type list.

```cpp
apply<type_constructor_t<T>, elements_t<T>> is T
```

Any meta-programming utilities working with type lists is out of the scope of this proposal, and so `meta::apply` is not proposed yet.

As `elements_t` has only a sense once we have a good definition of type list. This type trait is not proposed yet.

### Algorithms

While *Meta*, *Boost.MPL*, [MP11], [Brigand], and *Boost.Hana* define a lot of algorithms, these libraries have a
different approach. Meta defines them only for concrete types. Boost.MPL defines them following the STL run-time design, [MP11],[Brigand] define them for templates and Boost.Hana defines them following the function programming paradigm.

We believe that we need to decide of a direction from the committee. Nevertheless the authors consider that we need to define the algorithms based on meta-requirements of the types as Boost.MPL does, but based on the functional paradigm as Boost.Hana do. Most of the algorithms defined in Meta have a generalization once we find the good concept.

Design rationale

Why the meta-programming approach for C++2x?

Boost.Hana proposes to work with heterogeneous constexpr functions and to consider type as values in order to do meta-programming Boost.Hana-TypeComputations.

While the approach is a good one, the meta-programming syntax is not as friendly as the authors consider it is needed.

Compare

```
typename decltype(hana::partial(type_c<Fn>, type_c<Args>...))::type
```

to

```
meta::bind_front<Fn, Args...>
```

It is also true that this kind of expressions are only needed in Hana when you need to declare a type in function of other types.

It is also true that with type deduction, we don't need very often this kind of expressions. Maybe meta could be built on top of Boost.Hana design.

Why meta namespace?

We use the nested namespace meta to avoid conflicts with other names used already in std as invoke and is_invocable.

There will be also conflict with other meta utilities that will be proposed later on as list, apply.

An alternative could be to prefix them with the meta prefix, for example.
Another alternative is to have the nested namespace `meta` and introduce in `std` the aliases that we consider are the most useful and that don't have naming issues. This proposal doesn't goes yet in this direction.

Why `type_constructor`?

Having access to the type constructor allows to base some operations on the type constructor instead of in the type itself.

Examples of operation that work well with type constructors are for example `none<TC>() / make<TC>(v)`.

Why `placeholder::_t`?

We can define the type constructors using any name. However the current proposal has a `type_constructor<quote<Tmpl>>` specialization that consists in applying the template to the `placeholder::_t`.

Removing this specialization would mean that the user will need to specialize for example `type_constructor<quote<optional>>`.

About `rebind` and Allocator?

The standard provides already something similar for Allocator via the `A::template rebind<T>::other` expression.

`rebind_t` uses the nested type `type` instead of `other` as allocators does. This is done for coherency purposes. However, this would mean that `Allocators` are not `Rebindable`.

Impact on the standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++14.

Proposed wording

The proposed changes are expressed as edits to N4564 the Working Draft - C++ Extensions for Library Fundamentals.

General utilities library
20.10.x Header synopsis

namespace std
{
namespace experimental
{
inline namespace fundamental_v3
{
namespace meta
{

// Type alias for T::type
template <class T>
    using eval = typename T::type;

// Variable alias for T::value
template<class T>
    using eval_v = T::value;

// identity meta-function
template <class T>
    struct id {
        using type = T;
    };

template <class T>
    using id_t = eval<id, T>;

// Invocables

// invoke a type constructor TC with the arguments Xs
template<class TC, class... Xs>
    using invoke = typename TC::template invoke<Xs...>;

// Meta-function class
template <class TC>
    struct is_type_constructor;

template <class TC>
    constexpr bool is_type_constructor_v = is_type_constructor<TC>::value;

template <class, R = void>
    struct is_invocable; // not defined
    template <class Fn, class ...Args, class R>
```
struct is_invocable<Fn(Arg...), R>;

template <class Sig, R = void>
  constexpr bool is_invocable_v = is_invocable<Sig, R>::value;

// invokes a type constructor TC with the arguments Xs
template<class TC, class TL>
  using apply;

// identity Meta-Invocables
struct identity
{
  template <class T>
    using invoke = T;
};

// constant Meta-Invocables that returns always its argument T
template <class T>
  struct always
  {
    template <class...>
      using invoke = T;
  }

// Compose the Meta-Invocables Fs.
template <class ...Fs>
  struct compose;

// lifts a class template to a Meta-Invocables
template <template <class ...> class Tmpl>
  struct quote
  {
    template <class... Xs>
      using invoke = Tmpl<Xs...>;
  }

// lifts a type trait to a Meta-Invocables
template <template <class ...> class Trait>
  using quote_trait = compose<quote<eval>, quote<Trait>>;

// An Meta-Invocables that partially applies the Meta-Invocables F by binding the
// template <class F, class... Args>
  struct bind_front
  {
    template <class... Xs>
      using invoke = invoke<F, Args..., Xs...>;
  }
```
// An Meta-Invocables that partially applies the Meta-Invocables F by binding the
template <class F, class... Args>
struct bind_back
{
    template <class... Xs>
    using invoke = invoke<F, Xs..., Args...>;
};

//
template <class M, class ...U>
struct rebind : id<typename M::template rebind<U...>> {};

template <template<class ...> class TC, class ...Ts, class ...Us>
struct rebind<TC<Ts...>, Us...> : id<TC<Us...>> {};

template <class M, class ...Us>
using rebind_t = eval<rebind<M, Us...>>;

inline namespace placeholders
{
    // regular placeholders:
    struct _t {};
}

// Type Constructor trait
template <class T>
struct type_constructor;
template <template <class...> class Tmpl >
struct type_constructor<meta::quote<Tmpl>> : type_constructor<Tmpl<_t>> {};

template <class T>
using type_constructor_t = eval<type_constructor<T>>;
}
```cpp
template <class T>
struct id;
```

**Condition**

Always `T`.

**Preconditions**

`T` shall be a complete type.

**Template**

```cpp
template <class TC>
struct is_type_constructor;
```

**Condition**

If `TC::template invoke` is well formed then `true` else `false`.

**Preconditions**

`TC` shall be a complete type.

**Template**

```cpp
template <class, R = void>
struct is_invocable; // not defined
template <class TC, class ...Xs, class R>
struct is_invocable<TC<Xs...>, R>;
```

**Condition**

- If `TC::template invoke<Xs...>` is well formed then
  - If `R` is void `std::true_type`
  - Else `std::is_convertible<meta::invoke_t<Xs...>, R>`
  - Else `std::false_type`.

**Preconditions**

`TC` and all types in the parameter pack `Xs` shall be a complete types.

**Template**
template <class ...Fs>
struct compose;

Condition

The definition must satisfy

\begin{align*}
\text{invoke<compose<>}, \text{Ts}..\rangle & \text{ is ill-formed} \\
\text{is_same<invoke<compose<F>, \text{Ts}..\rangle, \text{invoke<F, \text{Ts}..\rangle}} & \\
\text{is_same<invoke<compose<F, \text{Fs}..\rangle, \text{Ts}..\rangle, \text{invoke<F, invoke<compose<\text{Fs}..\rangle, \text{Ts}..\rangle}} &
\end{align*}

Definition

\begin{align*}
\text{template <typename \ldots Fs>} \\
\text{struct compose} \\
\{ \\
\}; \\
\text{template <typename F>} \\
\text{struct compose<F>} \\
\{ \\
\quad \text{template <typename \ldots Ts>} \\
\quad \text{using invoke = invoke<F, \text{Ts}..\rangle}; \\
\}; \\
\text{template <typename F0, typename \ldots Fs>} \\
\text{struct compose<F0, Fs\ldots>} \\
\{ \\
\quad \text{template <typename \ldots Ts>} \\
\quad \text{using invoke = invoke<F0, invoke<compose<\text{Fs}..\rangle, \text{Ts}..\rangle\rangle}; \\
\};
\end{align*}

Preconditions

Fs shall be a complete types.

Template

\begin{align*}
\text{template <class T>} \\
\text{struct type_constructor;}
\end{align*}

Condition
If \( T::\text{type}_\text{constructor} \) is well formed then \( \text{id}<T::\text{type}_\text{constructor}> \).

**Preconditions**

\( T \) shall be a complete types.

**Remarks**

This template can be specialized by the user.

**Example of customizations**

Next follows some examples of customizations that could be included in the standard

```cpp
namespace std {
    namespace experimental {

        // Holder specialization
        template <>
        struct optional<_t>::meta::quote<optional> {};

    }
}
```

See [P0323R2](https://github.com/boostorg/preprocessor/blob/master/boost/preprocessor/detail/detail.p267959082a52737835a3a17aa0e217f1c0339831).
namespace std {
namespace experimental {

    // Holder specialization
    template <class E>
    struct expected<_t, E> : meta::bind_back<quote<expected>, E> {};

namespace meta {
    template <class T, class E>
    struct type_constructor<expected<T, E>> : id<expected<_t, E>> {};
}
}
}

future / shared_future

namespace std {

    // Holder specializations
    template <>
    struct future<experimental::_t> : experimental::meta::quote<future> {};
    template <>
    struct future<experimental::_t>&;
    template <>
    struct shared_future<experimental::_t> : experimental::meta::quote<shared_future> {};
    template <>
    struct shared_future<experimental::_t>&;

namespace experimental {
namespace meta {

    // type_constructor customization
    template <class T>
    struct type_constructor<future<T>> : id<future<_t>> {};
    template <class T>
    struct type_constructor<future<T>> : id<future<_t>>& {};
    template <class T>
    struct type_constructor<shared_future<T>> : id<shared_future<_t>> {};
    template <class T>
    struct type_constructor<shared_future<T>>& {};

}]]}

unique_ptr

namespace std {

    // Holder customization
    template <class D>
    struct unique_ptr<experimental::_t, D> {
        template <class ...T>
            using invoke = unique_ptr<T...,
                experimental::meta::eval<experimental::meta::rebind<D, T...>>;}

    namespace experimental {
    namespace meta {

        template <class T, class D>
        struct type_constructor<unique_ptr<T,D>> : meta::id<unique_ptr<T,D>> {}
        }}

}

shared_ptr

namespace std {
    // Holder customization
    template <>
    struct shared_ptr<experimental::_t> : experimental::meta::quote<shared_ptr> {
    }

}

pair
namespace std {

    // Holder customization
    template <>
    struct pair<experimental::_t, experimental::_t>
    {
        template <class ...Ts>
        using invoke = pair<Ts...>;
    };

namespace experimental {
namespace meta {

    // type_constructor customization
    template <class T1, class T2>
    struct type_constructor<pair<T1,T2>> : meta::id<pair<_t, _t>> {};

    template <>
    struct type_constructor<meta::quote<pair>> : meta::id<pair<_t, _t>> {};
}
}
}

namespace std {

    // Holder customization
    template <>
    struct tuple<experimental::_t> : experimental::meta::quote<tuple> { };

namespace experimental {
namespace meta {

    // type_constructor customization
    template <class ...Ts>
    struct type_constructor<tuple<Ts...>> : meta::id<tuple<_t>> {};
}
}
}

**Implementability**

This proposal can be implemented as pure library extension, without any compiler magic support, in C++14.

There is an implementation at
Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

- Should this be part of the Fundamentals TS or a separated Meta TS?
- Should the namespace `meta` be used for the meta programming utilities?
- Do we want nested template alias or nested type trait for `invoke`?
- Do we want the nested to be named `invoke` or `apply`?
- Is there an interest on `is_invocable`?
- Is there an interest on `is_type_constructor`?
- Is there an interest on placeholder type `_t`?
- Should the type constructors for `pair`, `tuple`, `optional`, `future`, `unique_ptr`, `shared_ptr` be part of this proposal? As specializations using the placeholder `_t` or with a suffix `_tc`?
- Is there an interest on `id`, `eval`?
- Is there an interest on `identity`?
- Is there an interest on `compose`?
- Is there an interest on `bind_front`, `bind_back`?
- Is there an interest on `quote`, `trait_quote`?
- Is there an interest on `rebind`?
- If yes, should `rebind` define a nested type alias `type` or a nested type alias `other` as allocators does?
- Is there an interest on `type_constructor`?

Future work

Add Meta-Product concept

Boost.Hana defines a `Product` as a type that allows to get the `first(t)` and `second(t)`.

We believe that a `Meta-Product` could be generalized to any number of arguments. `Meta-Product` can be seen as a subset of `Product` types that don't require to have a value.

We believe that `product_type::element<PT, N>` and `product_type::size<PT>` could be appropriated.

Add Meta-Foldable concept
Boost.Hana defines a Foldable.

We believe that a Meta-Foldable should require fold_left.

## Add Meta-Sequence concept

Boost.Hana defines a Sequence as a refinement of Iterable and Foldable.

Add a type `meta::list` as a model of Meta-Product and Meta-Sequence

## Add more Meta-Invocables related operations

- `meta::flip` -> MetaInvocable
- `meta::arg<size_t>` -> MetaInvocable
- `meta::elements<MetaInvocable>` -> Meta-Product

## Add algorithms on Meta-Products

- `meta::apply<MetaInvocable, MetaProduct>` -> Type
- `meta::front<MetaProduct>` -> Type
- `meta::back<MetaProduct>` -> Type
- `meta::is_empty<MetaProduct>` -> bool_constant

## Add algorithms on Meta-Foldable

- `meta::fold_right<MetaInvocable, Type, MetaFoldable>` -> Type
- `meta::apply<MetaInvocable, MetaFoldable>` -> Type
- `meta::for_each<MetaFoldable, MetaInvocable>` -> MetaFoldable
- `meta::size<MetaFoldable>` -> size_constant

## Add algorithms on Meta-Sequence

- `meta::size<MetaFoldable>

## Add lambdas
Acknowledgements

Many thanks to Eric Nibbler for his Meta library and Louis Idionne for his Boost.Hana library, which have both been used as inspiration of this proposal.

References

- **N4564** - Programming Languages — C++ Extensions for Library Fundamentals, Version 2 PDTS
  

- **P0196R3** Generic `none()` factories for Nullable types
  

- **P0323R2** - A proposal to add a utility class to represent expected monad (Revision 4)
  

- **P0327R2** Product types access
  

- **P0338R2** - A `make` factory
  

- Meta
  
  [https://github.com/ericniebler/meta](https://github.com/ericniebler/meta)

- Boost.Hana
  
  [https://github.com/boostorg/hana](https://github.com/boostorg/hana)

- Boost.Hana-Metafunction
  
  [http://boostorg.github.io/hana/group__group-Metafunction.html](http://boostorg.github.io/hana/group__group-Metafunction.html)

- Boost.Hana-TypeComputations
  
  [http://boostorg.github.io/hana/index.html#tutorial-integral](http://boostorg.github.io/hana/index.html#tutorial-integral)

- Boost.MPL
  
  [https://github.com/boostorg/mpl](https://github.com/boostorg/mpl)
TinyMeta Tiny Metaprogramming Library