Document number:	P0343R0
Date:	2016-05-22
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

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 P0338R0 and P0196R1 and other have been used in their respective implementations.

Table of Contents

Introduction

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 P0338R0 and P0196R1 and other have been used in their respective implementations.

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 defines also *Callable* in function of `std::invoke`, so we will use here *Meta-Callable*.

One of the uses of *Meta-Callabless* as type constructors as any *Meta-Callabless* return in some way a type. We call the 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.

P0196R1 and P0338R0 depends on this proposal.

Proposal

Type-Traits helpers

`meta::id`

Results always its parameter. This is useful to convert a type into a type trait.

Meta provides the same.

Boost.MPL call this `mpl::identity`.

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

`meta::eval`

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

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

Meta-Callables types

Requirements

A *Meta-Callable* 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 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

`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 call this `mpl::apply`.

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

Basic operations

`meta::identity`

Results always its `invoke` parameter.

Meta not supported to the author knowledge.

Boost.MPL call this `mpl::identity`.

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

`meta::always`

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

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

Boost.MPL call this `mpl::always`.

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

`meta::compose`

Composes several *Meta-Callabless*.

Meta provides the same.

Boost.MPL not supported to the author knowledge.

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

partial application

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`

Other *Meta-Callable* factories

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

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

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_callable<Fn(Args...), R>`

Checks if the result of invoking the class `T` with the arguments `Args...` is convertible to `R`.

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

Meta not supported to the author knowledge.

Boost.MPL not supported to the author knowledge.

Boost.Hana not supported to the author knowledge.

`meta::is_type_constructor`

Sometime we don't have yet the arguments to invoke the *Meta-Callable*, but we want to check that art 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_callable`.

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

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-Callables* that can be used to construct the type `T`

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

TypeConstructible Product types

Given that a *Product types* P0327R0 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>`

```
invoke<type_constructor<T>, product_type::element_t<T,1>, ..., product_type::element
```

Rebindable types

When we have a type, it is often useful to rebind 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

```
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

```
invoke<type_constructor_t<T>, Xs...>
```

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.

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 an *Meta-Callables* to the elements of a type list.

```
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 both Meta, Boost.MPL and Boost.Hana defines 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 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 `hana`.

Why `meta` namespace?

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

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 prefix `meta_` 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`.

This means 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 `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++1z.

Proposed wording

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

General utilities library

--- Insert a new section or include in 20.10.2 **--**

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 {
```

```cpp
    using type = T;
  };

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

// Callables

// 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_callable; // not defined
template <class Fn, class ...Args, class R>
  struct is_callable<Fn(Args...), R>;

template <class Sig, R = void>
  constexpr bool is_callable_v = is_callable <Sig, R>::value;

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

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

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

// Compose the Meta-Callabless Fs.
template <class ...Fs>
```

```
  struct compose;

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

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

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

// An Meta-Callables that partially applies the Meta-Callables F by binding the ar
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>>;

}
}
}
}
```

Change 20.10.6 [meta.rel], Table 51 — Type relationship predicates, add new rows with the following content: ---

Template

```
template <class T>
struct id;
```

Condition

Always `T`.

Preconditions

`T` shall be a complete type.

Template

```
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_callable; // not defined
template <class TC, class ...Xs, class R>
struct is_callable<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

```cpp
  template <class ...Fs>
  struct compose;
```

Condition

The definition must satisfy

```
invoke<compose<>, Ts..>> is ill-formmed
is_same<invoke<compose<F>, Ts..>>, invoke<F, Ts..>>
is_same<invoke<compose<F, Fs...>, Ts..>>, invoke<F, invoke<compose<Fs...>, Ts...>>>
```

Definition

```cpp
template <typename... Fs>
struct compose
{
};

template <typename F>
struct compose<F>
{
    template <typename... Ts>
    using invoke = invoke<F, Ts...>;
};

template <typename F0, typename... Fs>
struct compose<F0, Fs...>
{
    template <typename... Ts>
    using invoke = invoke<F0, invoke<compose<Fs...>, Ts...>>;
};
```

Preconditions

`Fs` shall be a complete types.

Template

```cpp
template <class T>
struct type_constructor;
```

Condition

If `T::type_constructor` is well formed then `id<TC::type_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

`optional`

```
namespace std {
namespace experimental {

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

}
}
```

`expected`

See P0323R0.

```
namespace std {
namespace experimental {

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

}}}
```

`tuple`

```
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

`https://github.com/viboes/std-make/include/experimental/meta.hpp`.

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_callable` ?
- 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 appropiated.

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-Callables* related operations

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

Add algorithms on *Meta-Products*

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

Add algorithms on *Meta-Foldable*

- `meta::fold_right<MetaCallable, Type, MetaFoldable> -> Type`
- `meta::apply<MetaCallable, MetaFoldable> -> Type`
- `meta::for_each<MetaFoldable, MetaCallable> -> 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

  http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4564.pdf

- P0196R1 Generic `none()` factories for *Nullable* types

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0196r1.pdf

- P0323R0 - A proposal to add a utility class to represent expected monad (Revision 2)

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0323r0.pdf

- P0327R0 Product types access

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0327r0.pdf

- P0338R0 - A `make` factory

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0338r0.pdf

- Meta

  https://github.com/ericniebler/meta

- Boost.Hana

  https://github.com/boostorg/hana

- Boost.Hana-Metafunction

  http://boostorg.github.io/hana/group__group-Metafunction.html

- Boost.Hana-TypeComputations

  http://boostorg.github.io/hana/index.html#tutorial-integral

- Boost.MPL Boost.MPL

  https://github.com/boostorg/mpl

- TinyMeta Tiny Metaprogramming Library

  http://ericniebler.com/2014/11/13/tiny-metaprogramming-library/