Document number:	P0650R0
Date:	2017-06-15
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escribá <vicente.botet@nokia.com>

C++ Monadic interface

Abstract

This paper proposes to add the following type of classes with the associated customization points and some algorithms that work well with them.

- *Functor*,
- *Applicative*
- *Monad*
- *Monad-Error*

This paper concentrates on the basic operations. More will come later if the committee accept the design (See Future Work section).

Table of Contents

History

Revision 0

Creation in response to request of the committee to split the expected proposal P0323R0 into a expected class P0323R0 and a monadic interface (this document).

Introduction

Most of you know *Functor*, *Applicative*, *Monad* and *MonadError* from functional programming. The underlying types of the types modeling *Functor*, *Applicatives*, *Monad* and *MonadError* are homogeneous, that is, the functions have a single type.

In the following notation `[T]` stands for a type wrapping instances of a type `T`, possibly zero or `N` instances. `(T -> U)` stands for a function taking a `T` as parameter and returning a `U`.

```
functor::transform : [T] x (T->U) -> [U]
functor::map : (T->U) x [T] -> [U]

applicative::ap : [T] x [(T->U)] -> [U]
applicative::pure<A> : T -> [T]

monad::unit<A> : T -> [T]
monad::bind : [T] x (T->[U]) -> [U] //mbind
monad::flatten : [[T]] -> [T] // unwrap
monad::compose : (B->[C]) x (A->[B])-> (A->[C])

monad_error::make_error<M>: E -> error_type_t<M,E>
monad_error::catch_error: [T] x (E->T) -> [T] where E = error_type_t<[T]>
monad_error::catch_error: [T] x (E->[T]) -> [T]
```

Motivation and Scope

From Expected proposals

Adapted from P0323R0 taking in account the proposed non-member interface.

Safe division

This example shows how to define a safe divide operation checking for divide-by-zero conditions. Using exceptions, we might write something like this:

```
struct DivideByZero: public std::exception {...};
double safe_divide(double i, double j)
{
    if (j==0) throw DivideByZero();
    else return i / j;
}
```

With `expected<T,E>`, we are not required to use exceptions, we can use `std::error_condition` which is easier to introspect than `std::exception_ptr` if we want to use the error. For the purpose of this example, we use the following enumeration (the boilerplate code concerning `std::error_condition` is not shown):

```
enum class arithmetic_errc
{
    divide_by_zero, // 9/0 == ?
    not_integer_division // 5/2 == 2.5 (which is not an integer)
};
```

Using `expected<double, error_condition>` , the code becomes:

```
expected<double,error_condition> safe_divide(double i, double j)
{
    if (j==0) return make_unexpected(arithmetic_errc::divide_by_zero); // (1)
    else return i / j; // (2)
}
```

(1) The implicit conversion from `unexpected_type<E>` to `expected<T,E>` and (2) from `T` to `expected<T,E>` prevents using too much boilerplate code. The advantages are that we have a clean way to fail without using the exception machinery, and we can give precise information about why it failed as well. The liability is that this function is going to be tedious to use. For instance, the exception-based

```
function i + j/k is:
double f1(double i, double j, double k)
{
    return i + safe_divide(j,k);
}
```

but becomes using `expected<double, error_condition>` :

```
expected<double, error_condition> f1(double i, double j, double k)
{
    auto q = safe_divide(j, k)
    if(q) return i + *q;
    else return q;
}
```

This example clearly doesn't respect the "clean code" characteristic introduced above and the readability doesn't differ much from the "C return code". Hopefully, we can see `expected<T,E>` through functional glasses as a monad. The code is cleaner using the function `functor::transform` . This way, the error handling is not explicitly mentioned but we still know, thanks to the call to `transform` , that something is going underneath and thus it is not as silent as exception.

```
expected<double, error_condition> f1(double i, double j, double k)
{
    return functor::transform(safe_divide(j, k), [&](double q) {
        return i + q;
        });
}
```

The `transform` function calls the callable provided if expected contains a value, otherwise it forwards the error to the callee. Using lambda function might clutter the code, so here the same example using functor:

```
expected<double, error_condition> f1(double i, double j, double k)
{
    return functor::transform(safe_divide(j, k), bind(plus, i, _1));
}
```

We can use `expected<T, E>` to represent different error conditions. For instance, with integer division, we might want to fail if the two numbers are not evenly divisible as well as checking for division by zero. We can overload our `safe_divide` function accordingly:

```
expected<int, error_condition> safe_divide(int i, int j)
{
    if (j == 0) return make_unexpected(arithmetic_errc::divide_by_zero);
    if (i%j != 0) return make_unexpected(arithmetic_errc::not_integer_division);
    else return i / j;
}
```

Now we have a division function for integers that possibly fail in two ways. We continue with the exception oriented

```
function i/k + j/k:
int f2(int i, int j, int k)
{
    return safe_divide(i,k) + safe_divide(j,k);
}
```

Now let's write this code using an `expected<T,E>` type and the functional `transform` already used previously.

```
expected<int,error_condition> f(int i, int j, int k)
{
    return monad::bind(safe_divide(i, k), [=](int q1) {
        return functor::transform(safe_divide(j,k), [=](int q2) {
            return q1+q2;
        });
    });
}
```

The compiler will gently say he can convert an `expected<expected<int, error_condition>, error_condition>` to `expected<int, error_condition>`. This is because the function `functor::transform` wraps the result in `expected` and since we use twice the map member it wraps it twice. The function `monad::bind` (do not confound with `std::bind`) wraps the result of the continuation only if it is not already wrapped. The correct version is as follow:

```
expected<int, error_condition> f(int i, int j, int k)
{
    return monad::bind(safe_divide(i, k), [=](int q1) {
    return monad::bind(safe_divide(j,k), [=](int q2) {
    return q1+q2;
        });
    });
}
```

The error-handling code has completely disappeared but the lambda functions are a new source of noise, and this is even more important with `n` `expected` variables. Propositions for a better monadic experience are discussed in section [Do-Notation], the subject is left open and is considered out of scope of this proposal.

Error retrieval and correction

The major advantage of `expected<T,E>` over `optional<T>` is the ability to transport an error, but we didn't come yet to an example that retrieve the error. First of all, we should wonder what a programmer do when a function call returns an error:

1. Ignore it.
2. Delegate the responsibility of error handling to higher layer.
3. Trying to resolve the error.

Because the first behavior might lead to buggy application, we won't consider it in a first time. The handling is dependent of the underlying error type, we consider the `exception_ptr` and the `error_condition` types.

We spoke about how to use the value contained in the `expected` but didn't discuss yet the error usage.

A first imperative way to use our error is to simply extract it from the `expected` using the `error()` member function. The following example shows a `divide2` function that return `0` if the error is `divide_by_zero`:

```
expected<int, error_condition> divide2(int i, int j)
{
    auto e = safe_divide(i,j);
    if (!e && e.error().value() == arithmetic_errc::divide_by_zero) {
        return 0;
    }
    return e;
}
```

This imperative way is not entirely satisfactory since it suffers from the same disadvantages than `value()`.

Again, a functional view leads to a better solution. The `catch_error` member calls the continuation passed as argument if the expected is erroneous.

```
expected<int, error_condition> divide3(int i, int j)
{
    auto e = safe_divide(i,j);
    return monad_error::catch_error(e, [](const error_condition& e){
        if(e.value() == arithmetic_errc::divide_by_zero)
        {
            return 0;
        }
        return make_unexpected(e);
    });
}
```

An advantage of this version is to be coherent with the `bind` and `map` member functions. It also provides a more uniform way to analyze error and recover from some of these. Finally, it encourages the user to code its own "error-resolver" function and leads to a code with distinct treatment layers.

Proposal

This paper proposes to add the following type of classes with the associated customization points and the algorithms that work well with them.

- *Functor*,
- *Applicative*
- *Monad*
- *Monad-Error*

These are the basic operations. More will come later if the committee adopt the design (See Future Work section).

Design Rationale

Most of the design problems for this library are related to the names, signatures and how to this type of classes are customized. See [CUSTOM] for a description of an alternative approach to customization points. This proposal is based on this alternative approach, but it could be adapted to other approaches once we decide which is the mechanism we want to use.

Functor

`transform` versus `fmap`

The signature of the more C++ `transform` function is different from the usual *Functor* `map` function.

```
transform : [T] x (T->U) -> [U]
map : (T->U) x [T] -> [U]
```

`transform` has the advantage to be closer to the STL signature.

The advantage of the `map` is that it can be extended to a variadic number of *Functors*.

```
map : (T1x...<Tn->U) x [T1] x ... x [Tn]-> [U]
```

Both seem to be useful, and so we propose both in this paper.

Applicative

`ap` ?

`pure`

We don't define an additional `applicative::pure` function as we have already `type_constructible::make` P0338R1.

Monad

`unit`

We don't define an additional `monad::unit` function as we have already `type_constructible::make` P0338R1.

`bind`

C++ has the advantage to be able to overload on the parameters of the signature.

`bind` can be overloaded with functions that return a *Monad* or functions that return the *ValueType* as it proposed for `std::experimental::future::then`.

The authors don't see any inconvenient in this overload, but would prefer to have an additional function that supports this ability, so that we know that chain will only work with functions that return a *Monad*.

Note that the user could use `transform` and `bind` to get this overload.

`bind` function parameter parameter

The `bind` function accepts functions that take the `ValueType` as parameter. `std::experimental::future::then` function parameter takes a `future<T>`.

We could define `bind` in function of a possibly `then` function (or whatever is the appropriated name) when the type provides access to the `ValueType` as it is the case for `std::future` and all the *Nullable* types. However the authors don't know how to do it in the general case.

`monad::unwrap`

For the time being we don't propose to have `monad::unwrap` as a customization point.

We define it in function of `monad::bind`.

`monad::compose`

Customization: ADL versus traits

These concepts have some functions that cannot be customized using overload (ADL), as the dispatching type is not a function parameters,e.g. `pure<TC>(C)` or `make_error<TC>(E)`.

We have also some customization points that are types, as `error_type<T>::type`

The authors prefer to have a single customization mechanism, and traits is the one that allows to do everything.

Boost.Hana uses a similar mechanism.

See [CUSTOM] where we describe the advantages and liabilities of each approach.

Customization: All at once or one by one

Boost.Hana has chosen to customize each operation individually. The approach of this proposal is closer to how other languages have addressed the problem, that is, customize all operations at once.

There are advantages and liabilities to both approaches. See [CUSTOM] where we describe the advantages and liabilities of each approach.

Allow alternative way to customize a type of classes

Some type of classes can be customized using different customization points. E.g. *Monad* can be customized by either defining `bind` or `flatten`. The other customization point being defined in function of the other.

This proposal uses an emulation to what Haskel calls minimal complete definition, that is a struct that defines some operations given the user has customized other.

About names

There is a tradition in functional languages as Haskell with names that could be not the more appropriated for C++.

`functor::map` alternatives

We have already a clear meaning of `map` in the standard library, the associative ordered container `std::map`? The proposed `functor::map` function is in scope `std::experimental::functor::map` so there is no possible confusion. Haskell uses `fmap` instead of `functor::map` as it has no namespaces, but we have them in C++. [Boost.Hana] doesn't provides it.

`applicative::pure` versus `type_constructible::make`

Haskell uses `pure` as factory of an applicative functor. The standard library uses `make_` for factories. In addition we have already the proposed `type_constructible::make` [P0338R1] that plays the same role.

Boost.Hana uses `lift`. However [Boost.Hana] provides also a Core `make` facility not associated to any concept.

`applicative::ap` versus `applicative::apply`

`monad::unit` versus `type_constructible::make`

`monad::bind` versus `monad::chain`

We have already a clear meaning of `bind` in the standard library function `std::bind`, which could be deprecated in a future as we have now lambda expressions. The proposed `bind` (Haskell uses `mbind`) is in scope `std::experimental::monad::bind` so there is no possible confusion. [Boost.Hana] uses `chain` instead. [Boost.Hana] locates all the function isn namespace `boost::hana`.

`monad::flatten` versus `monad::unwrap` versus `monad::join`

[THEN] original proposal had a `future::unwrap` function that unwraps a wrapped `future`. Haskell uses `join`. Boost.Hana uses `flatten`.

`monad_error::throw_error` versus `monad_error::make_error`

Haskell uses `throw_error` as factory for monaderror errors. If we choose `make` to wrap a value, it seems coherent to use `make_error` instead of throwerror as C++ has exceptions. We are not throwing an error but building it.We have the proposed `type_constructible::make` [P0338R1] that plays the same role.

Operators

Language based do-notation

Customization

This paper is based on an alternative customization approach [CUSTOM]. While this is not an imperative this approach helps to define such concepts.

Factory functions

Both *Applicative* and *Monad* have factory function `applicative::pure` and `monad::unit`. We have already such a factory function isolated in the *Factory* concept via `type_constructible::make`.

We could define those specific factory functions but functions that forward to the `factory::make` function, but there is not too much interest in duplicating such factories. We can just nest the factory namespace in `applicative` meaning that any `Applicative` is a `Factory`.

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++17. There are however some classes in the standard that need to be customized.

This paper depends in some way on the helper classes proposed in P0343R0, as e.g. the place holder `_t` and the associated specialization for the type constructors `optional<_t>`, `unique_ptr<_t>`, `shared_ptr<_t>`.

Proposed Wording

The proposed changes are expressed as edits to N4564 the Working Draft - C++ Extensions for Library Fundamentals V2.

Add a "Functor Types" section

Functor Types

Functor requirements

A *Functor* is a type constructor that supports the `transform` function. A type constructor `TC` meets the requirements of *Functor* if:

- `TC` is a *TypeConstructor*
- for any T *EqualityComparable DefaultConstructible*, and *Destructible*, `invoke_t<TC,T>` satisfies the requirements of *EqualityComparable DefaultConstructible*, and *Destructible*,
- the expressions shown in the table below are valid and have the indicated semantics, and
- `TC` satisfies all the other requirements of this sub-clause.

In Table X below, `t` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a rvalue of type `F` where `F` satisfies *Callable*.

Expression	Return Type	Operational Semantics
invoke_t<TC, VT...>	T	
type_constructor_t<T>	TC	
functor::transform(t, f)	invoke_t<TC,U>	Applies \`f\` to the contents of \`t\`.

Header synopsis [functor.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
namespace functor {

  struct tag{};

  // class traits
  template <class TC, class Enabler=void>
    struct traits {};

  template <class T, class F>
    `see below` transform(T&& x, F&& f);
  template <class T, class P, class F>
    `see below` adjust_if(T&& x, P&& p, F&& f);

  struct mcd_transform;
  struct mcd_adjust_if;

}

  template <class T>
    struct is_functor;
  template <class T>
    inline constexpr bool is_functor_v = is_functor<T>::value;
  template <class T>
    struct is_functor<const T> : is_functor<T> {};
  template <class T>
    struct is_functor<volatile T> : is_functor<T> {};
  template <class T>
    struct is_functor<const volatile T> : is_functor<T> {};
}
}
}
```

Class `tag` [functor.tag]

A tag used to identify the *functor* types. The customization of the `traits<T>` must inherit from this tag.

Class Template `traits` [functor.traits]

```
namespace functor {
    template <class T, class Enabler=void>
        struct traits {};
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

Function Template `transform` [functor.transform]

```cpp
namespace functor {
  template <class T, class F>
    auto transform(T&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<T>>`

Effects: forward the call to the `traits<TC>::transform`

Remark: The previous function shall not participate in overload resolution unless:

- `T` has a type constructor `TC` that satisfies *Functor*,
- `F` is a *Callable* taking as parameter the `ValueType` of `T` and result `U`,
- The result of `transform` is the rebinding of `T` with the result of the invocation of `f` with the value of `x`.

```
transform : [T] x T->U -> [U]
```

Function Template `adjust_if` [functor.adjust_if]

```cpp
namespace functor {
  template <class T, class P, class F>
    auto adjust_if(T&& x, P&& p, F&& f);
}
```

Let `TC` be `type_constructor<decay_t<T>>`

Effects: forward the call to the `traits<TC>::adjust_if`

Remark: The previous function shall not participate in overload resolution unless:

- `T` has a type constructor `TC` that satisfies *Functor*,
- `F` is a *Callable* taking as parameter the `ValueType` of `T` and result `U`,
- `P` is a *Predicate* taking as parameter the `ValueType` of `T`,
- The result of `adjust_if` is the rebinding of `T` with the result of the invocation of `f` with the value of `x`.

```
adjust_if : [T] x T->bool x T->U -> [U]
```

class `mcd_transform` [functor.mcd_transform]

```cpp
namespace functor {
  struct mcd_transform : functor::tag
  {
  template <class T, class P, class F>
    auto adjust_if(T&& x, P&& p, F&& f);
  };
}
```

This minimal complete definition defines `adjust_if` in function of `transform`.

class `mcd_transform:: adjust_if` [functor.mcdtransform.adjustif]

```
namespace functor {
  template <class T, class P, class F>
    auto mcd_transform::adjust_if(T&& x, P&& p, F&& f);
}
```

Equivalent to:

```
functor::transform(x, [&](auto x) { if pred(x) then return f(x) else return x; });
```

class `mcd_adjust_if` [functor.mcdadjustif]

```
namespace functor {
  struct mcd_adjust_if : functor::tag
  {
  template <class T, class F>
    auto transform(T&& x, F&& f);
  };
}
```

This minimal complete definition define `transform` in function of `adjust_if`.

class `mcd_adjust_if::transform` [functor.mcdadjustif.transform]

```
namespace functor {
  template <class T, class F>
    auto mcd_adjust_if::transform(T&& x, F&& f);
}
```

Equivalent to:

```
functor::adjust_if(x, always(true), f);
```

where `always(true)` is a function object that return always `true`.

Template class `is_functor` [functor.is_functor]

```
  template <class T>
    struct is_functor : is_base_of<functor::tag, functor::traits<T>> {};
```

Add a "Applicative Types" section

Applicative Functor Types

Applicative requirements

A *Applicative* is a type constructor that supports the *Functor* requirements and supports the `ap` function. A type constructor `TC` meets the requirements of *Applicative* if:

- `TC` is a *Functor*
- the expressions shown in the table below are valid and have the indicated semantics, and
- `TC` satisfies all the other requirements of this sub-clause.

In Table X below, `a` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a rvalue of type `invoke<TC,T>` where `F` satisfies *Callable*.

Expression	Return Type	Operational Semantics
invoke_t<TC, VT...>	T	
type_constructor_t<T>	TC	
applicative::ap(a, f)	rebind_t<TC,U>	Applies the contents of `f` to the contents of `a`.

Header synopsis [functor.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
namespace applicative {
    using namespace functor;

  struct tag{};

  // class traits
  template <class TC, class Enabler=void>
    struct traits {};

  template <class A, class F>
    `see below` ap(A&& x, F&& f);
}

  template <class T>
    struct is_applicative;
  template <class T>
    inline constexpr bool is_applicative_v = is_applicative<T>::value;
  template <class T>
    struct is_applicative<const T> : is_applicative<T> {};
  template <class T>
    struct is_applicative<volatile T> : is_applicative<T> {};
  template <class T>
    struct is_applicative<const volatile T> : is_applicative<T> {};
}
}
}
```

Class `tag` [applicative.tag]

A tag used to identify the *applicative* types. The customization of the `traits<T>` must inherit from this tag.

Class Template `traits` [functor.traits]

```
namespace functor {
    template <class T, class Enabler=void>
        struct traits {};
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

Function Template `ap` [applicative.ap]

```
namespace applicative {
  template <class A, class F>
    auto ap(A&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<A>>`

Effects: forward the call to the `traits<TC>::ap`.

Remark: The previous function shall not participate in overload resolution unless:

- `A` has a type constructor `TC` that satisfies *Applicative*,
- `F` has a type constructor `TC` that satisfies *Applicative*,
- `value_type_t<F>` is a *Callable* taking as parameter the `ValueType` of `T` and result `U`,
- The result of `ap` is the rebinding of `T` with the result of the invocation of the contents of `f` with the value of `x`.

```
ap : [T] x [T->U] -> [U]
```

Template class `is_applicative` [applicative.is_applicative]

```
  template <class T>
    struct is_applicative : is_base_of<applicative::tag, applicative::traits<T>> {};
```

Add a "Monad Types" section

Monad Types

Monad requirements

A *Monad* is a type constructor that in addition to supporting *Applicative* supports the `bind` function. A type constructor `TC` meets the requirements of *Monad* if:

- `TC` is an *TypeConstructor*
- for any T *EqualityComparable DefaultConstructible*, and *Destructible*, `invoke_t<TC,T>` satisfies the requirements of *EqualityComparable DefaultConstructible*, and *Destructible*,
- the expressions shown in the table below are valid and have the indicated semantics, and
- `TC` satisfies all the other requirements of this sub-clause.

In Table X below, `m` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a *Callable* rvalue of type `F`. In Table X below, `m` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a rvalue of type `F` where `F` satisfies *Callable(T)>*.

Expression	Return Type	Operational Semantics
invoke_t<TC, VT...>	T	
type_constructor_t<T>	TC	
monad::bind(m, f)	invoke_t<TC,U>	Applies `f` to the contents of `m`.

Header synopsis [monad.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
namespace monad {
  using namespace applicative;

  struct tag{};

  // class traits
  template <class TC, class Enabler=void>
    struct traits {};

  template <class T, class F>
    `see below` bind(T&& x, F&& f);
  template <class T>
    `see below` unwrap(T&& x);

  struct mcd_bind;
  struct mcd_unwrap;


}

  template <class T>
    struct is_monad;
  template <class T>
    inline constexpr bool is_monad_v = is_monad <T>::value;
  template <class T>
    struct is_monad<const T> : is_monad<T> {};
  template <class T>
    struct is_monad<volatile T> : is_monad<T> {};
  template <class T>
    struct is_monad<const volatile T> : is_monad<T> {};
}
}
}
```

Class `tag` [monad.tag]

A tag used to identify the *monad* types. The customization of the `traits<T>` must inherit from this tag.

Class Template `traits` [monad.traits]

```cpp
namespace monad {
    template <class T, class Enabler=void>
        struct traits {};
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

Function Template `transform` [monad.bind]

```cpp
namespace monad {
  template <class M, class F>
    auto bind(M&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<M>>`
Let `T` be `value_type<decay_t<M>>`

Effects: forward the call to the `traits<TC>::bind`. This function must return the result of calling to the `f` parameter with the contained value type, if any; Otherwise it must return a monad of the same type that `F` returns without a value type.

Remark: The previous function shall not participate in overload resolution unless:

- `M` has a type constructor `TC` that satisfies *monad*,
- `F` satisfies `Callable<F, invoke_t<TC,U>(T)>` where `T` is the `ValueType` of `M` for some type `U`,
- The result of `bind` is the result of the invocation of `f` with the value of `x` if any, otherwise an `invoke_t<TC,U>(T)` instance without a value.

```
bind : [T] x T->[U] -> [U]
```

Function Template `unwrap` [monad.unwrap]

```cpp
namespace monad {
  template <class M>
    auto unwrap(M&& x)
}
```

Let `TC` be `type_constructor<decay_t<M>>`

Effects: forward the call to the `traits<TC>::unwrap`. This function should flatten input monad on a Monad that has one less nested level.

Remark: The previous function shall not participate in overload resolution unless:

- `M` has a type constructor `TC` that satisfies *monad*,
- `M` has the form form `TC<TC<T>>` where `T` is `value_type_t<value_type_t<decay_t<M>>>`
- The result of `unwrap` is the monad `TC<T>`.

```
unwrap : [[T]] -> [T]
```

Class `mcd_bind` [monad.mcd_bind]

```
namespace functor {
  struct mcd_bind : monad::tag
  {
  template <class T>
    auto unwrap(T&& x);
  };
}
```

This minimal complete definition define `unwrap` in function of `bind`.

Class `mcd_bind::unwrap` [monad.mcd_bind.unwrap]

```
namespace functor {
  template <class T>
    auto mcd_bind::unwrap(T&& x);
}
```

Equivalent to:

```
monad::bind(x, idnetity, f);
```

where `identity` is a unary function object that return its parameter.

Class `mcd_unwrap` [monad.mcd_unwrap]

```
namespace functor {
  struct mcd_unwrap : monad::tag
  {
    template <class T, class F>
        auto bind(T&& x, F&& f);
  };
}
```

This minimal complete definition defines `bind` in function of `unwrap` and `transform`.

Class `mcd_unwrap::bind` [monas.mcd_unwrap.bind]

```
namespace functor {
  template <class T, class F>
    auto mcd_unwrap(T&& x, F&& f);
}
```

Equivalent to:

```
monad::unwrap(functor::transform(x, f));
```

Template class `is_monad` [monad.is_monad]

```
template <class T>
  struct is_monad : is_base_of<monad::tag, monad::traits<T>> {};
```

Add a "Monad Error Types" section

Monad Error Types

MonadError requirements

A *MonadError* is a type constructor that in addition to supporting *Monad* supports the `make_error` and the `catch_error` functions. A type constructor `TC` meets the requirements of *MonadError* if:

- `TC` is an *Monad*
- the expressions shown in the table below are valid and have the indicated semantics, and
- `TC` satisfies all the other requirements of this sub-clause.

In Table X below, `m` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a *Callable* rvalue of type `F`. In Table X below, `m` denotes an rvalue of type `invoke<TC,T>`, `f` denotes a rvalue of type `F` where `F` satisfies *Callable(T)>*.

Expression	Return Type	Operational Semantics
invoke_t<TC, VT...>	T	
type_constructor_t<T>	TC	
error_type_t<TC>	E	
monad_error::make_error(e)	Err	a instance of a type depending on error_type_t<TC> that is convertible to any invoke_t.
monad_error::catch_error(m, f)	M	Applies f to the error of m if any. Otherwise it return m.

Header synopsis [monad_error.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v3 {
namespace monad_error {
  using namespace monad;

  struct tag{};

  // class traits
  template <class TC, class Enabler=void>
    struct traits {};

  template <class M>
  struct error_type
  {
    using type = typename traits<M>::template error_type<M>;
  };

  template <class M>
  using error_type_t = typename error_type<M>::type;

  template <class T, class F>
    `see below` catch_error(T&& x, F&& f);
}

  template <class T>
    struct is_monad_error;
  template <class T>
    inline constexpr bool is_monad_error_v = is_monad_error<T>::value;
  template <class T>
    struct is_monad_error<const T> : is_monad_error<T> {};
  template <class T>
    struct is_monad_error<volatile T> : is_monad_error<T> {};
  template <class T>
    struct is_monad_error<const volatile T> : is_monad_error<T> {};
}
}
}
```

Class `tag` [monad.tag]

A tag used to identify the *monad* types. The customization of the `traits<T>` must inherit from this tag.

Class Template `traits` [monad.traits]

```
namespace monad {
    template <class T, class Enabler=void>
        struct traits {};
}
```

Remark The `Enabler` parameter is a way to allow conditional specializations.

Function Template `catch_error` [monaderror.catcherror]

```
namespace monad {
  template <class M, class F>
    auto catch_error(M&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<M>>`
Let `T` be `value_type<decay_t<M>>`
Let `E` be `error_type<decay_t<M>>`

Effects: forward the call to the `traits<TC>::catch_error` . This function must return the result of calling to the `f` parameter with the contained error type, if any; Otherwise it must returns the parameter `x` .

Remark: The previous function shall not participate in overload resolution unless:

- `M` has a type constructor `TC` that satisfies *monad*,
- `F` satisfies `Callable<F, M(E)>` where `E` is the `ErrorType` of `M` ,
- The result of `catch_error` is the result of the invocation of `f` with the error of `x` if any, otherwise `x` .

```
catch_error : [T]:E x E->[T] -> [T]:E
```

Function Template `recover` [monad_error.recover]

```
namespace monad {
  template <class M, class F>
    auto recover(M&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<M>>`
Let `T` be `value_type<decay_t<M>>`
Let `E` be `error_type<decay_t<M>>`

Effects: forward the call to the `traits<TC>::catch_error` . This function must return the result of calling to the `f` parameter with the contained error type, if any; Otherwise it must returns the parameter `x` .

Remark: The previous function shall not participate in overload resolution unless:

- `M` has a type constructor `TC` that satisfies *monad*,
- `F` satisfies `Callable<F, T(E)>` where `E` is the `ErrorType` of `M` and `T` is the value type of `M` ,
- The result of `recover` is the result of the invocation of `f` with the error of `x` wrapped on a `M` if any, otherwise `x` .

```
recover : [T]:E x E->T -> [T]:E
```

Function Template `adapt_error` [monaderror.adapterror]

```
namespace monad {
  template <class M, class F>
    auto adapt_error(M&& x, F&& f)
}
```

Let `TC` be `type_constructor<decay_t<M>>`

Let `T` be `value_type<decay_t<M>>`
Let `E` be `error_type<decay_t<M>>`

Effects: forward the call to the `traits<TC>::catch_error`. This function must return the result of calling to the `f` parameter with the contained error type, if any; Otherwise it must returns the parameter `x`.

Remark: The previous function shall not participate in overload resolution unless:

- `M` has a type constructor `TC` that satisfies *monad*,
- `F` satisfies `Callable<F, G(E)>` where `E` is the `ErrorType` of `M` and `G` is another error type,
- The result of `adapt_error` is the result of the invocation of `f` with the error of `x` wrapped on a `TC` if any, otherwise the value wrapped with `TC`.

```
catch_error : [T]:E x E->G -> [T]:G
```

Template class `is_monad_error` [monaderror.ismonad_error]

```

  template <class T>
    struct is_monad_error : is_base_of<monad_error::tag, monad_error::traits<T>> {};
```

Customization for *Nullable* Types

Add Specializations of *Functor*, *Applicative*, *Monad* and *MonadError*.

Nullable objects can be seen as *Functor*, *Applicative*, *Monad* and *MonadError*.

```
namespace nullable {
  template <class N, class F>
  constexpr `see below` transform(N&& n, F&& f);

  template <class N, class F
  constexpr `see below` ap(F&& f, N&& n);

  template <class M, class F>
  constexpr `see below`
  bind(M&& m, F&& f);

  template <class M>
  constexpr `see below`
  make_error();

  template <class M, class F>
  constexpr `see below`
  catch_error(M&& m, F&& f);

  struct as_functor: functor::tag  {
    template <class T, class F>
      static constexpr auto transform(T&& x, F&& f) {
        return nullable::transform(forward<T>(x), forward<F>(f));
      }
  };
  struct as_applicative: applicative::tag {
```

```
    template <class T, class F>
      static constexpr auto ap(F&& f, T&& x) {
        return nullable::ap(forward<F>(f), forward<T>(x));
      }
  };

  struct as_monad: monad::tag {
    template <class M, class F>
      static constexpr auto bind(M&& x, F&& f) {
        return nullable::bind(forward<M>(x), forward<F>(f));
      }
  };
  struct as_monad_error: monad_error::tag {
    template <class M, class F>
      static constexpr auto catch_error(M&& x, F&& f) {
        return nullable::catch_error(forward<M>(x), forward<F>(f));
      }
  };
}
namespace functor {
    template <class N>
    struct traits<N, meta::when<
        is_nullable<N>::value && is_type_constructible<N>::value
    >> : nullable::as_functor {};
}
namespace applicative {
    template <class N>
    struct traits<N, meta::when<
        is_nullable<N>::value && is_type_constructible<N>::value
    >> : nullable::as_applicative {};
}

namespace monad {
    template <class N>
    struct traits<N, meta::when<
        is_nullable<N>::value && is_type_constructible<N>::value
    >> : nullable::as_monad {};
}

namespace monad_error {
    template <class N>
    struct traits<N, meta::when<
        is_nullable<N>::value && is_type_constructible<N>::value
    >> : nullable::as_monad_error {};
}
```

Customization for Expected Objects

Add Specialization of *expected* [expeced.object.monadic_spec].

```
namespace functor {
  template <class T, class E>
  struct traits<expected<T,E>> : functor::tag
  {
    template <class Expected, class F>
      static constexpr auto transform(Expected&& x, F&& f);
  };
}
namespace applicative {
    template <class T, class E>
    struct traits<expected<T,E>> : applicative::tag
    {
        template <class Expected, class F>
        static auto ap(F&& f, Expected&& x);
    };
}
namespace monad {
    template <class T, class E>
    struct traits<expected<T,E>> : monad::tag
    {
        template <class M, class F>
        static constexpr auto bind(M&& x, F&& f);
    };
}
namespace monad_error {
    template <class T, class E>
    struct traits<expected<T,E>> : monad_error::tag
    {
        template <class M>
        using error_type = typename M::error_type;

        template <class M, class ...Xs>
        static constexpr auto make_error(Xs&& ...xs);

        template <class M, class F>
        static constexpr auto catch_error(M&& x, F&& f);
    };
}
```

Implementability

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

Open points

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

- Do we want the proposed customization approach?
- Do we want separated proposals for each type class?
- Should a smart pointer (a pointer) be considered a *Functor*?
- Should a *Nullable* be considered a *MonadError*?
- Should `std::array`, `std::vector` be considered a *Functor*?
- Should a *Product Type* be considered a *Functor*, *Applicative*, *Monad* when all the elements have the same type?

Future work

Add more algorithms

Based on what Boost.Hana and [Boost.Fusion] provides already, extend the basic functionality with useful algorithms.

Functor algorithms

```
functor::adjust : [T] x CT x (T->U) -> [U]
functor::fill : [T] x U -> [U]
functor::replace_if : [T] x (T->bool) x T -> [T]
functor::replace : [T] x CT x T -> [T]
```

Applicative algorithms

```
applicative::lift : [T] x (T->bool) x (T->U) -> [U]
```

Monad algorithms

```
monad::then : [[T]] -> [T] // do
monad::next : [T] x ([T]->U) -> [U] // then
monad::next : [T] x ([T]->[U]) -> [U]
```

Add *N-Functor*, *N-Applicative*, *N-Monad*

Do we need *N-Functor*, *N-Applicative*, *N-Monad* that support *Product Type*?

Add Transformers

Monadic type don't compose very well. We need some kind of transformer that facilitates their composition. See `Haskell Transformers`.

See how to add `Haskell::Alternative` type class

Add *Monoids* and *MonadPlus* type classes

Add *Foldable* type classes

Acknowledgements

Thanks to Louis for his work on the monadic interface of Boost.Hana.

Special thanks and recognition goes to Technical Center of Nokia - Lannion for supporting in part the production of this proposal.

References

- Boost.Hana Boost.Hana library

  http://boostorg.github.io/hana/index.html

- N4564 N4564 - Working Draft, C++ Extensions for Library Fundamentals, Version 2 PDTS

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4564.pdf

- P0088R0 Variant: a type-safe union that is rarely invalid (v5)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0088r0.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

- P0323R1 A proposal to add a utility class to represent expected monad (Revision 3)

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0323r1.pdf

- P0338R1 C++ generic factories

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0338r1.pdf

- P0343R0 - Meta-programming High-Order functions

  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0343r0.pdf

- [SUM_TYPE] Generic Sum Types