Expected: Exception-friendly MonadError

C++Now 2014

Vicente Botet Escriba, Pierre Talbot
vicente.botet@wanadoo.fr/ptalbot@hyc.io

Alcatel-Lucent International-Lannion/University of Pierre et Marie Curie-Paris (France)

2014, May 16

Content

About Pierre Talbot

- Belgian (French side).
- Still a student, at the University Pierre et Marie Curie in Paris.
- Interest in software engineering, language design and concurrency.

Open-source involvement

- **Boost C++ library** Working on Boost.Check and Boost.Expected
- **Wesnoth** The add-on server (UMCD).

About Vicente Botet Escriba

Alcatel·Lucent

- Spanish.
- Work for Alcatel-Lucent International in Lannion - France.
- Interest in software engineering, language design, concurrency and STM.

Open-source involvement

- **Boost C++ library** Co-author and maintainer of Boost.Ratio, Boost.Chrono, Boost.Thread and some utilities in Boost.Utility.
- **Toward Boost C++ library** author of a lot of unfinished prototypes.

About Boost.Expected

- Based on the original idea of Andrei Alexandrescu (Systematic Error Handling in C++).
- Interface based on the new std :: experimental :: optional<T> type,taking advantage of the new C++11/14 features.
- Extended to conform to the MonadError concept.
- The project born as a proposal the authors made for GSoC 2013.
- We are currently working on a C++ standard proposal that we expect to be ready for Rapperswil.
- Once the interface would be more stable we will propose it for review to Boost.

- Git repository : https://github.com/ptal/Boost.Expected
- C++ draft proposal : http://www.hyc.io/boost/expected-proposal.pdf

Overview

expected<E, T> answer to the question "do you contain a value of type T or an error of type E?".

```
template <class E, class T>
class expected {
  bool has_value_;
  union{
    T val_;
    E err_;
  };
};
```

- expected<E,T> = T + unexpected_type<E>
- T != unexpected_type<E>
- We want that expected<E,T> behaves as T.
- expected<std::exception_ptr, T> = Expected<T>

How to create an expected object

- Valued.

```
expected<exception_ptr, int> ei = make_expected(1);
expected<exception_ptr, int> ej{2}; // alternative syntax
expected<exception_ptr, int> ek = ej; // ek has value 2

expected<exception_ptr,MoveOnly> el{in_place, "arg"};
expected<exception_ptr,MoveOnly> el{expect, "arg"};
  // calls MoveOnly{"arg"} in place
```

How to create an unexpected object

- Unexpected.

```
expected<errc, int> ek = make_unexpected(errc::invalid);

expected<string,int> em{unexpect, "arg"};
  // unexpected value, calls string{"arg"} in place

expected<string,int> e;  // e is unexpected...
ei = {}; // reset to unexpected
```

Observing an expected object

- Explicit conversion to bool.
- operator*() narrow contract

```
if (expected<exception_ptr, char> ch = readNextChar()) {
  process_char(*ch);
}
```

- value() wide contract

```
expected<errc,int> ei = str2int(s);
try {
    process_int(ei.value());
}  catch(bad_expected_access const&) {
    std::cout << "this was not a number";
}
```

Content

safe_divide exception-based style

```cpp
struct DivideByZero: public std::exception {...};
```

```cpp
int safe_divide(int i, int j)
{
  if (j==0) throw DivideByZero();
  else return i / j;
}
```

safe_divide Expected-based style

```
1 Expected<int> safe_divide(int i, int j)
2 {
3   if (j==0) return Expected<int>::from_error(DivideByZero());
4   else return Expected<int>(i / j);
5 }
```

- (3,4) redondant use of `Expected<int>`.

safe_divide expected-based style

Using boost::expected<exception_ptr, int> to carry the error condition.

```
expected<exception_ptr,int> safe_divide(int i, int j)
{
  if (j==0) return make_unexpected(DivideByZero());
  else return i / j;
}
```

- (3) uses implicit conversion from unexpected_type<E> to expected<E, T>.
- (4) uses implicit conversion from T to expected<E, T>.

- 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.

safe_divide expected-style - getting the value or an exception

- The caller obtains the stored value using the member value() function.

```
auto x = safe_divide(i,j).value(); // throw on error
```

- The caller can also store the expected value and check if a value is stored using the bool conversion function.

```
auto x = safe_divide(i,j); // won't throw
if (! x)
  int i = x.value(); // just "re"throw the stored exception
```

i + j/k expected-style - getting the value when available

- When the user knows that there is a value it can use the narrowed contract operator*().

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

i + j/k - functional style

```
expected<exception_ptr,int> f1(int i, int j, int k)
{
  return safe_divide(j, k).fmap([i](int q){return i + q;});
}
```

fmap calls the provided function if expected contains a value and wraps the result, otherwise it forwards the error to the callee.
Alternative using an existing functor.

```
expected<exception_ptr,int> f1(int i, int j, int k)
{
  return safe_divide(j, k).fmap(bind(add, i, _1)));
}
```

safe_divide - exception-based Multiple error conditions

```
struct NotDivisible: public std::exception {
  int i, j;
  NotDivisible(int i, int j) : i(i), j(j) {}
};

int safe_divide(int i, int j) {
  if (j == 0) throw DivideByZero();
  if (i%j != 0) throw NotDivisible(i,j);
  else return i / j;
}
```

```
T divide(T i, T j) {
  try { return safe_divide(i,j) }
  catch(NotDivisible& ex) { return ex.i/ex.j; }
  catch(...) { throw; }
}
```

safe_divide - expected-based Multiple error conditions

```
expected<exception_ptr,int> safe_divide(int i, int j) {
  if (j == 0) return make_unexpected(DivideByZero());
  if (i%j != 0) return make_unexpected(NotDivisible(i,j));
  else return i / j;
}
```

```
expected<exception_ptr,int> divide(int i, int j) {
  auto e = safe_divide(i,j);
  if(e.has_exception<NotDivisible>()) return i/j;
  else return e;
}
```

safe_divide - expected-based Multiple error conditions

```
expected<exception_ptr,int> divide(int i, int j) {
  return safe_divide(i,j)
  .catch_exception<NotDivisible>([](auto &ex)
    return ex.i/ex.j;
  });
}
```

i/k + j/k - scale

- exception-based.

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

- expected-based.

```
expected<exception_ptr,int> f2(int i, int j, int k) {
  auto q1 = safe_divide(i, k);
  if (!q1) return make_unexpected(q1);

  auto q2 = safe_divide(j, k);
  if (!q2) return make_unexpected(q2);

  return *q1 + *q2;
}
```

i/k + j/k - scale

- expected-based functional style.

```
expected<exception_ptr,int> f(int i, int j, int k) {
  return safe_divide(i, k).mbind([=](int q1) {
      return safe_divide(j,k).fmap([=](int q2) {
        return q1+q2;
      });
    });
}
```

mbind calls the provided function if expected contains a value, otherwise it forwards the error to the callee.

$i/k + j/k$ - scale

```
auto add = [](int s1, int s2){ return s1+s2; };
expected<exception_ptr,int> f(int i, int j, int k){
  return fmap(add, safe_divide(i, k), safe_divide(j, k));
}
```

The fmap non-member function calls the provided function if all the expected contains a value and wraps the result, otherwise it forwards the first error to the callee.

language-like style - error propagation

Possible C++ language extension: Based on the Haskell do-expression.
Something similar to the proposed await expression for futures.

```
expected<exception_ptr,int> f2(int i, int j, int k) {
  return ( auto s1 <- safe_divide(i, k) :
           auto s2 <- safe_divide(j, k) :
           s1 + s2
  );
}
```

transformed in

```
expected<exception_ptr,int> f2(int i, int j, int k) {
  return mbind(safe_divide(i, k),[&](auto s1) {
    return mbind(safe_divide(j, k),[&](auto s2) {
      return s1 + s2;
    });
  });
}
```

i/k + j/k - DO-macro

```
expected<exception_ptr,int> f2(int i, int j, int k) {
  return DO (
    ( s1, safe_divide(i, k) )
    ( s2, safe_divide(j, k) )
    s1 + s2
  );
}
```

i/k + j/k - EXPECT-macro

```
#define EXPECT(V, EXPR) \
auto BOOST_JOIN(_expected_,V) = EXPR; \
if (! BOOST_JOIN(_expected_,V))
  return make_unexpected(BOOST_JOIN(_expected_,V).error()); \
auto V =*BOOST_JOIN(_expected_,V)
```

```
expected<exception_ptr,int> f2(int i, int j, int k) {
  EXPECT(s1, safe_divide(i, k) );
  EXPECT(s2, safe_divide(j, k));
  return s1 + s2;
}
```

getIntRange - exception based

Given

```
int getInt(istream_range& r);
void matchesString(std::string, istream_range& r);
```

```
pair<int,int> getIntRange(istream_range& r)
{
    auto f = getInt(r);
             matchesString("..", r);
    auto l = getInt(r);
    return make_pair(f, l);
}
```

getIntRange - expect based

Given

```
expected<errc, int> getInt(istream_range& r);
expected<errc, void> matchesString(istream_range& r);
```

```
expected<errc, pair<int, int>> getIntRange(istream_range& r)
{
  return
    auto  f <- getInt(r) :
             matchedString("..", r) :
    auto  l <- getInt(r) :
    std::make_pair(f, l);
}
```

Content

class unexpected_type<E>

Between explicit and implicit conversion.
Building an explicit type that is implicitly convertible to another type.
E.g. imaginary<T> is explicitly convertible from a T and complex<T> could be implicitly convertible from imaginary<T>.

```
template <class E=std::exception_ptr>
class unexpected_type {
public:
    unexpected_type() = delete;
    constexpr explicit unexpected_type(E e)
        : error_(e) {}
    constexpr E value() const { return error_; }
};
```

unexpected factories

```
template <class E>
constexpr unexpected_type<decay_t<E>> make_unexpected(E v)  {
    return unexpected_type<decay_t<E>> (ex);
}

unexpected_type<>
  make_unexpected_from_current_exception() {
    return unexpected_type<> (std::current_exception());
}

template <class T, class E>
constexpr
unexpected_type<E> make_unexpected(expected<E, T> v)  {
    return unexpected_type<E>(v.error());
}
```

expected<E, T> conversions from/to E and unexpected_type<E>

- expected<E, T> is not convertible from E .
- expected<E, T> is implicitly convertible from unexpected_type<E> .
- The opposite is not true. Needs explicit function get_unexpected().

```
template <class E, class T>
class expected {
  // ...
  constexpr expected(unexpected_type<E>&& e);
  expected&  operator=(unexpected_type<E>&& e);

  // ...
  constexpr E const& error() const& noexcept;
  constexpr E& error() & noexcept;
  constexpr E&& error() && noexcept;
  constexpr unexpected_type<E> get_unexpected() const&;
  constexpr unexpected_type<E> get_unexpected() &&;
};
```

expected<exception_ptr, T> conversions from/to unexpected_type<E>

- expected<exception_ptr, T> is implicitly convertible from unexpected_type<E> for any E.

```
template <class T>
class expected<exception_ptr, T> {
  // ...
  constexpr expected(unexpected_type<exception_ptr>&& e);
  expected& operator=(unexpected_type<exception_ptr>&& e);
  template <class E>
  constexpr expected(unexpected_type<E>&& e);
  template <class E>
  expected& operator=(unexpected_type<E>&& e);

  // ...
};
```

expected<E, T> conversions from/to T

- expected<E, T> is implicitly convertible from T .
- The opposite is not true. Needs explicit function value().

```
template <class E, class T>
class expected {
  // ...
  constexpr expected(T const& v);
  constexpr expected(T&& v);
  expected&  operator=(T&& v);

  constexpr T const& value() const&;
  constexpr T& value() &;
  constexpr T&& value() &&;

};
```

Default constructor

While expected<exception_ptr, T> is default constructible, there is no exception stored in.
Alternatives:

- value() is undefined behavior if there is no exception stored or
- it throws a specific exception if there is no exception stored.

pointer−like

```
template <class E, class T>
class expected {
  // ...
  constexpr explicit operator bool() const noexcept;

  constexpr T const& operator*() const& noexcept;
  constexpr T& operator*() & noexcept;
  constexpr T&& operator*() && noexcept;

  constexpr T const  * operator->() const noexcept;
  constexpr T* operator*() noexcept;

};
```

Monadic functions

	Object	From	To	Result
fmap	M<E,T>	T	U	M<E,U>
fmap	M<E,T>	T	M<E,U>	M<E,M<E,U>>
mbind	M<E,T>	T	U	M<E,U>
mbind	M<E,T>	T	M<E,U>	M<E,U>
catch_error	M<E,T>	E	T	M<E,T>
catch_error	M<E,T>	E	M<E,T>	M<E,T>
then	M<E,T>	M<E,T>	U	M<E,U>
then	M<E,T>	M<E,T>	M<E,U>	M<E,U>

fmap member

```
template <typename F>
constexpr expected<E, result_of_t<F(T)>>
expected<E,T>::fmap(F&& f,
  REQUIRES( ! is_void_v<result_of_t<F(T)>> )
) const
{
  typedef expected<E, result_of_t<F(T)>> result_type;
  if (*this) return result_type(f(**this));
  else       return get_unexpected();
}
```

mbind member

```
template <typename F>
constexpr expected<E, result_of_t<F(T)>>
expected<E,T>::mbind(F&& f,
  REQUIRES( ! is_void_v<result_of_t<F(T)>> )
) const
{
  typedef expected<E, result_of_t<F(T)>> result_type;
  if (*this) return f(**this);
  else       return get_unexpected();
}
```

catch_error member

```
template <typename F>
constexpr expected<E,T>
expected<E,T>::catch_error(F&& f,
  REQUIRES( is_same_v<result_of_t<F(E)>, expected<E,T>> )
) const
{
  if ( ! *this ) return f(error());
  else           return *this;
}
```

then

```
template <typename F>
constexpr result_of_t<F(T)>
expected<E,T>::then(F&& f,
  REQUIRES( is_expected_v<result_of_t<F(expected<E,T>)> > )
)
{
  f(move(*this));
}
```

Relational operators

- unexpected values are always less than expected ones.
- How unexpected values compare?
- std :: exception_ptr doesn't compare.
- Do we want expected<exception_ptr,T> be comparable?
- compare as optional<T>?
- If yes, all the unexpected_type<exception_ptr> compare equals.
- This seems counterintuitive, as the observable behavior is different.
- expected<E,T> is comparable if unexpected_type<E> and T are comparable.
- unexpected_type<E> is comparable if E is comparable and
- unexpected_type<std::exception_ptr> is not comparable.

Relational operators

```
template <class T, class E>
constexpr bool
operator<(const expected<E,T>& x, const expected<E,T>& y)
{
  return (x)
    ? (y) ? *x < *y : false
    : (y) ? true : x.get_unexpected()<y.get_unexpected();
}

template <class T, class E>
constexpr bool
operator==(const expected<E,T>& x, const expected<E,T>& y)
{
  return (x)
    ? (y) ? *x == *y : false
    : (y) ? false : x.get_unexpected()==y.get_unexpected();
}
```

expected factories

```
template<typename T>
constexpr expected<std::exception_ptr, decay_t<T> >
  make_expected(T&& v ) ;
expected<std::exception_ptr, void> make_expected();
template <typename T, typename E>
expected<std::exception_ptr, T>
  make_expected_from_exception(E e)
template <typename T, typename E>
constexpr expected<decay_t<E>, T>
  make_expected_from_error(E e);
```

```
auto e1 = make_expected(2); // expected<exception_ptr,int>
auto e2 = make_expected_from_exception<int>(bad_alloc());
// expected<exception_ptr,int>
auto e3 = make_expected_from_error<int>(errc::invalid);
// expected<errc,int>
```

expected<E> as a type constructor

```
auto e1 = make_expected<errc>(2); // compile fails
auto e1 = expected<errc,int>(2);  // int is redondant
auto e2 = expected<errc>::make(2);  // no redondant
```

```
template <typename E=std::exception_ptr, class T=holder>
class expected;
template <typename E>
class expected<E,holder> {
public:
  template <class T> using type = expected<E,T>;
  template <class T> expected<E,T> make(T&& v) {
    return expected<E,T>(std::forward(v));
  }
};
```

Differences between expected<E, T> and expected<exception_ptr, T>

	expected<E, T>	expected<exception_ptr, T>
never-empty warranty	if E	almost yes
relational operators	if E and T	no
hash	if E and T	no
has_exception	no	yes
catch_exception	no	yes

- Should expected<E,T> and expected<exception_ptr,T> be represented by two different classes?

boost :: variant <unexpected_type<E>,T> Comparison

	variant	**expected**
never-empty warranty	yes	almost yes
factories	no	yes
value_type	no	yes
default constructor	yes (if E is)	yes (if E is)
observers	get<T>/get<E>	operator*/value/error
visitation	apply_visitor	fmap/mbind/catch_error

std :: future<T> and expected<exception_ptr,T> Comparison

	future	**expected**
specific null value	no	no
relational operators	no	depends
factories	make_ready_future	make_expected
factories	no	make_unexpected
emplace	no	yes
value_type	no	yes
default constructor	yes(invalid)	yes (E())
state	valid / ready	operator bool
observers	get /wait	operator* / value / error
visitation	then	fmap/mbind/catch_error
grouping	when_all/when_any	n.a.

expected/future differences

- Should we add a make<future<>> as make<expected<E>>?
- Should we make future<T> be implicitly constructible from unexpected_type<exception_ptr>>?
- Should we add emplace functions for future<T>?
- Should we add nested value_type to future<T>?
- Should we add valid () and ready() functions to expected<E,T> that return always true?
- Should we add operator bool to future<T>?
- Should we add fmap()/mbind()/catch_error() functions to future<T>?
- Should we add value() to future<T>?
- Should we add error () to future<T>?

Conversions from/to ready std::experimental::future<T>

```
template <class T>
expected<exception_ptr,T> make_expected(future<T>&& f) {
  assert (f.ready() && "future not ready");
  try {
    return f.get();
  } catch (...) {
    return make_unexpected_from_exception();
  }
}
```

```
template <class T>
std::future<T>
make_ready_future(expected<exception_ptr, T>&& e) {
  if (e) return make_ready_future(*e);
  else return make_unexpected_future<T>(e.error());
}
```

make_unexpected_future

```
template <class T, class E>
std::future<T> make_unexpected_future(E e)  {
  std::promise<T> p;
  std::future<T> f = p.get_future();
  p.set_exception(std::make_exception_ptr(e));
  return std::move(f);
}
```

make_expected alternative implementation

- If future<T> stores the exception on a exception_ptr, could define this function as a friend function.

```
template <class T>
expected<exception_ptr, T> make_expected(future<T>&& f) {
  assert (f.ready() && "future not ready");
  lock_guard<mutex> lk(f.get_mutex());
  if (f.has_value(lk)) return f.get(lk);
  else return make_unexpected(f.get_exception_ptr(lk));
}
```

- If future<T> stores a expected<exception_ptr,T>.

```
template <class T>
expected<exception_ptr, T> make_expected(future<T>&& f) {
  return expected<exception_ptr, T>(f);
}
```

std::experimental::optional<T> Comparison

- expected<E, T> is as an std :: experimental :: optiona<T>l that collapse all the values of E to nullopt .
- expected<nullopt_t, T> should behave as much as possible as optional <T>.

	optional	**expected**
specific null value	yes	no
relational operators	yes	depends
swap	yes	yes
factories	make_optional	make_expected
value_type	no	yes
default constructor	yes(nullopt)	yes (E())
observers	value	value / error
unwrap	no	yes
visitation	no	fmap/mbind/catch_error

expected/optional Differences

- Should we add a make<optional<>> as make<expected<E>>?
- Should we make expected<E,T> be implicitly constructible from E>?
- Should we add nested value_type to optional<T>?
- Should we add fmap()/mbind()/catch_error() functions to optional<T>?
- Should we add error () to optional<T>?
- Should we add unwrap() to optional<T>?

std::experimental::optional<T> Conversions

```
template <class T>
optional<T> make_optional(expected<E, T> v) {
  if (v) return make_optional(*v);
  else nullopt;
}
```

```
struct conversion_from_nullopt {};
template <class T>
expected<exception_ptr, T> make_expected(optional<T> v) {
  if (v) return make_expected(*v);
  else make_unexpected(conversion_from_nullopt());
}
```

Content

Open Points ...

- Should the operator== collapse all the unexpected values?
- Should expected<errc,T> and expected<exception_ptr,T> be represented by two different classes?
- Should expected<E,T> throw E instead of bad_expected_access?
- Should expected<E,T> be convertible from E when it is not convertible to T?
- Should we have operators for fmap()/mbind()/catch_error() (^/&/|)?
- Should fmap()/mbind()/catch_error() expect that the continuation doesn't throws?

Future Work ...

- Allocator support for expected.
- Define a common visitation interface for any, variant<E1, ..., En>, exception_ptr.
- Define an Error concept that would make expected<Error,T> more generic.

Conclusions

- expected<E, T> monadic functions are useful tools,
- that allows to combine functions that return expected<E, T> but,
- it would be much easier to use it with a specific language do-expression.

- expected<E, T>, future<T> and optional<T> share a lot of things but have some differences.
- Defining a common interface for the functions that have the same behavior allows us to define generic algorithms on top of these concepts.
- Having a monadic common interface would be one step towards this goal.

Thanks for your attention!

Questions?

Infix operators

```
auto e2 = e1 ^ f1 ^ f2;
```

```
auto e2 = e1 ^fmap^ f1 ^fmap^ f2;
```

```
return ( e1 <- f1 : e2 <- f2 : e1+e2 ) | rec;
```

```
return ( e1 <- f1 : e2 <- f2 : e1+e2 ) ^catch_error^ rec;
```