C++ METAPROGRAMMING

A PARADIGM SHIFT

Louis Dionne, C++Now 2015

QUADRANTS OF COMPUTATION IN C++

RUNTIME COMPUTATIONS (CLASSIC)

RUNTIME SEQUENCES

```
std::vector<int> ints{1, 2, 3, 4};
```

RUNTIME FUNCTIONS

```
std::string f(int i) {
  return std::to_string(i * i);
}

std::string nine = f(3);
```

RUNTIME ALGORITHMS

```
std::vector<std::string> strings;
std::transform(ints.begin(), ints.end(),
               std::back_inserter(strings), f);
```

`constexpr` COMPUTATIONS

constexpr SEQUENCES

```
constexpr std::array<int, 4> ints{1, 2, 3, 4};
```

`constexpr` FUNCTIONS

```
constexpr int factorial(int n) {
  return n == 0 ? 1 : n * factorial(n - 1);
}

constexpr int six = factorial(3);
```

constexpr ALGORITHMS

```
template <typename T, std::size_t N, typename F>
  constexpr std::array<std::result_of_t<F(T)>, N>
transform(std::array<T, N> array, F f) {
  // ...
}

constexpr std::array<int, 4> ints{1, 2, 3, 4};
constexpr std::array<int, 4> facts = transform(ints, factorial);
```

HETEROGENEOUS COMPUTATIONS (FUSION)

HETEROGENEOUS SEQUENCES

```
fusion::vector<int, std::string, float> seq{1, "abc", 3.4f};
```

HETEROGENEOUS FUNCTIONS

```
struct to_string {
  template <typename T>
  std::string operator()(T t) const {
    std::stringstream ss;
    ss << t;
    return ss.str();
  }
};

std::string three = to_string{}(3);
std::string pi = to_string{}(3.14159);
```

HETEROGENEOUS ALGORITHMS

```
fusion::vector<int, std::string, float> seq{1, "abc", 3.4f};
fusion::vector<std::string, std::string, std::string> strings =
                fusion::transform(seq, to_string{});
```

TYPE-LEVEL COMPUTATIONS (MPL)

TYPE-LEVEL SEQUENCES

```
using seq = mpl::vector<int, char, float, void>;
```

TYPE-LEVEL FUNCTIONS

```
template <typename T>
struct add_const_pointer {
  using type = T const*;
};

using result = add_const_pointer<int>::type;
```

TYPE-LEVEL ALGORITHMS

```
using seq = mpl::vector<int, char, float, void>;
using pointers = mpl::transform<seq,
                     mpl::quote1<add_const_pointer>>::type;
```

NOTE

```
fusion::vector != mpl::vector
```

CONSIDER

```
mpl::vector<int, void, char>{} // ok

// error: field has incomplete type void
fusion::vector<int, void, char>{}
```

CLAIM

TYPE-LEVEL $\subset$ HETEROGENEOUS

WHY BOTHER?

C++14 IS A SERIOUS GAME CHANGER

→ WE CAN DO BETTER NOW

SEE FOR YOURSELF

CHECKING FOR A MEMBER: THEN

```
template <typename T, typename = decltype(&T::xxx)>
static std::true_type has_xxx_impl(int);

template <typename T>
static std::false_type has_xxx_impl(...);

template <typename T>
struct has_xxx
  : decltype(has_xxx_impl<T>(int{}))
{ };

struct Foo { int xxx; };
static_assert(has_xxx<Foo>::value, "");
```

CHECKING FOR A MEMBER: SOON

```
template <typename T, typename = void>
struct has_xxx
  : std::false_type
{ };

template <typename T>
struct has_xxx<T, std::void_t<decltype(&T::xxx)>>
  : std::true_type
{ };

struct Foo { int xxx; };
static_assert(has_xxx<Foo>::value, "");
```

CHECKING FOR A MEMBER: WHAT IT SHOULD BE

```
auto has_xxx = is_valid([](auto t) -> decltype(t.xxx) {});

struct Foo { int xxx; };
Foo foo{1};

static_assert(has_xxx(foo), "");
static_assert(!has_xxx("abcdef"), "");
```

INTROSPECTION: THEN

```
namespace keys {
  struct name;
  struct age;
}

BOOST_FUSION_DEFINE_ASSOC_STRUCT(
  /* global scope */, Person,
  (std::string, name, keys::name)
  (int, age, keys::age)
)

int main() {
    Person john{"John", 30};
    std::string name = at_key<keys::name>(john);
    int age = at_key<keys::age>(john);
}
```

INTROSPECTION: NOW

```
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (int, age)
  );
};

int main() {
  Person john{"John", 30};
  std::string name = at_key(john, BOOST_HANA_STRING("name"));
  int age = at_key(john, BOOST_HANA_STRING("age"));
}
```

GENERATING JSON: THEN

```
// sorry, not going to implement this
```

GENERATING JSON: NOW

```
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (int, age)
  );
};

Person joe{"Joe", 30};
std::cout << to_json(make_tuple(1, 'c', joe));
```

Output:

```
[1, "c", {"name" : "Joe", "age" : 30}]
```

HANDLE BASE TYPES

```
std::string quote(std::string s) { return "\"" + s + "\""; }

template <typename T>
auto to_json(T const& x) -> decltype(std::to_string(x)) {
  return std::to_string(x);
}

std::string to_json(char c) { return quote({c}); }
std::string to_json(std::string s) { return quote(s); }
```

HANDLE Sequences

```
template <typename Xs>
  std::enable_if_t<models<Sequence, Xs>(),
std::string> to_json(Xs const& xs) {
  auto json = transform(xs, [](auto const& x) {
    return to_json(x);
  });

  return "[" + join(std::move(json), ", ") + "]";
}
```

HANDLE Structs

```
template <typename T>
  std::enable_if_t<models<Struct, T>(),
std::string> to_json(T const& x) {
  auto json = transform(keys(x), [&](auto name) {
    auto const& member = at_key(x, name);
    return quote(to<char const*>(name)) + " : " + to_json(member);
  });

  return "{" + join(std::move(json), ", ") + "}";
}
```

STILL NOT CONVINCED?

HERE'S MORE

ERROR MESSAGES: THEN

```
using xs = mpl::reverse<mpl::int_<1>>::type;
```

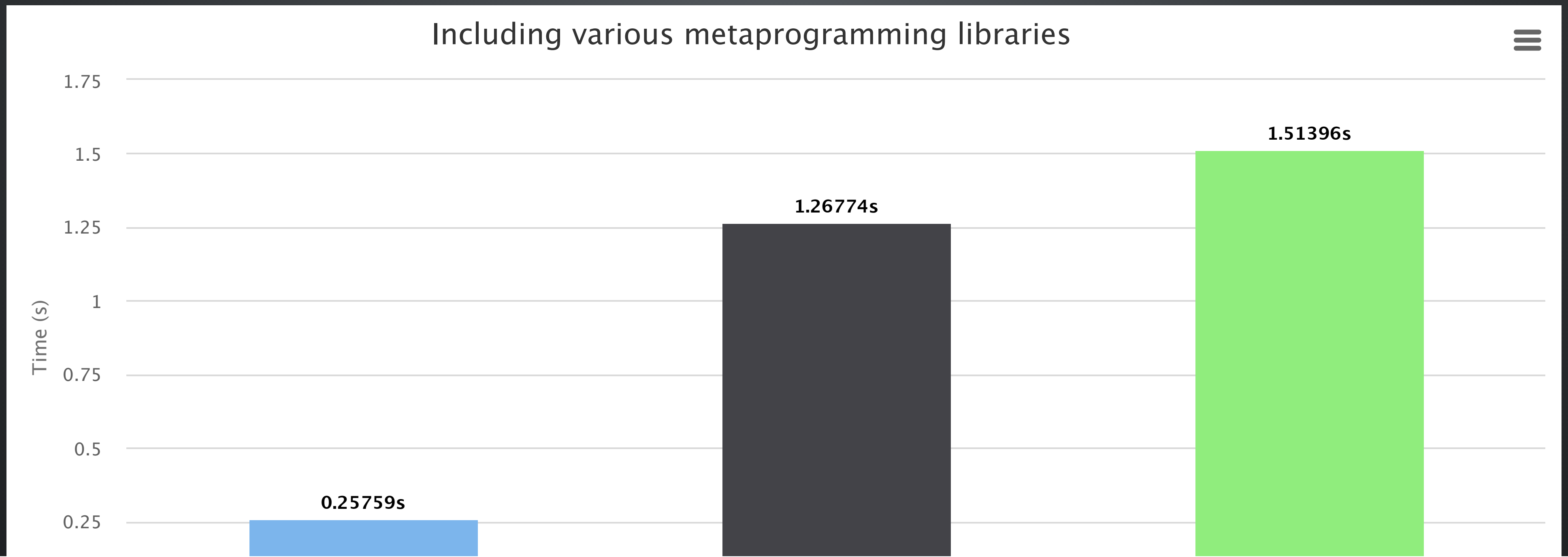
```
boost/mpl/clear.hpp:29:7: error:
  implicit instantiation of undefined template
  'boost::mpl::clear_impl<mpl_::integral_c_tag>::apply<mpl_::int_<1> >'
    : clear_impl< typename sequence_tag<Sequence>::type >
      ^
```

ERROR MESSAGES: NOW

```
auto xs = hana::reverse(1);
```

```
boost/hana/fwd/sequence.hpp:602:13: error:
  static_assert failed "hana::reverse(xs) requires xs to be a Sequence"
        static_assert(_models<Sequence, typename datatype<Xs>::type>{},
        ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error_messages.cpp:19:24:
  in instantiation of function template specialization
  'boost::hana::_reverse::operator()<int>' requested here
auto xs = hana::reverse(1);
                       ^
```

COMPILE-TIMES: THEN AND NOW



0

Hana Boost.MPL Boost.Fusion

Highcharts.com

Compile-time behavior of transform

Time (s)

2.5
2
1.5
1
0.5
0

0 50 100 150 200 250 300 350 400

- hana::tuple
- hana::tuple_t with hana::Metafunction
- mpl::vector
- fusion::vector

Number of elements

Highcharts.com

WE MUST RETHINK METAPROGRAMMING

BUT HOW?

HERE'S MY TAKE

WHAT IS AN `integral_constant`?

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant type;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
};
```

A TYPE-LEVEL ENCODING OF A `constexpr` VALUE

```
using one = integral_constant<int, 1>;
```

A `constexpr`-PRESERVING OBJECT

```
template <typename T>
void f(T t) { static_assert(t == 1, ""); }

constexpr int one = 1;
f(one);
// error: t is not constexpr in the body of f

f(integral_constant<int, one>{});
// ok: implicit conversion to int is constexpr
```

WE CAN DO `constexpr` ARITHMETIC

```
constexpr int two = ((3 * 2) + 4) / 5;
```

integral_constant VERSION

```
template <typename X, typename Y>
struct plus {
  using type = integral_constant<
    decltype(X::value + Y::value),
    X::value + Y::value
  >;
};

using three = plus<integral_constant<int, 1>,
                   integral_constant<int, 2>>::type;
```

BUT WHAT IF?

```
template <typename V, V v, typename U, U u>
constexpr auto
operator+(integral_constant<V, v>, integral_constant<U, u>)
{ return integral_constant<decltype(v + u), v + u>{}; }

template <typename V, V v, typename U, U u>
constexpr auto
operator==(integral_constant<V, v>, integral_constant<U, u>)
{ return integral_constant<bool, v == u>{}; }

// ...
```

TADAM!

```
static_assert(decltype(
  integral_constant<int, 1>{} + integral_constant<int, 4>{}
                             ==
                integral_constant<int, 5>{}
  )::value, "");
```

(OR SIMPLY)

```
static_assert(integral_constant<int, 1>{} + integral_constant<int, 4>{}
                                                    ==
                                    integral_constant<int, 5>{}
, "");
```

PASS ME THE SUGAR, PLEASE

```
template <int i>
constexpr integral_constant<int, i> int_{};

static_assert(int_<1> + int_<4> == int_<5>, "");
```

MORE SUGAR

```
template <char ...c>
constexpr auto operator"" _c() {
  // parse the characters and return an integral_constant
}

static_assert(1_c + 4_c == 5_c, "");
```

EUCLIDEAN DISTANCE

$$\text{distance}\left((x_1, y_1), (x_2, y_2)\right) := \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)$$

COMPILE-TIME ARITHMETIC: THEN

```
template <typename P1, typename P2>
struct distance {
  using xs = typename minus<typename P1::x,
                            typename P2::x>::type;
  using ys = typename minus<typename P1::y,
                            typename P2::y>::type;
  using type = typename sqrt<
    typename plus<
      typename multiplies<xs, xs>::type,
      typename multiplies<ys, ys>::type
    >::type
  >::type;
};

static_assert(equal_to<
  distance<point<int_<3>, int_<5>>, point<int_<7>, int_<2>>>::type,
  int_<5>
>::value, "");
```

COMPILE-TIME ARITHMETIC: NOW

```
template <typename P1, typename P2>
constexpr auto distance(P1 p1, P2 p2) {
  auto xs = p1.x - p2.x;
  auto ys = p1.y - p2.y;
  return sqrt(xs*xs + ys*ys);
}

static_assert(distance(point(3_c, 5_c), point(7_c, 2_c)) == 5_c, "");
```

BUT RUNTIME ARITHMETIC WORKS TOO

```
auto p1 = point(3, 5); // dynamic values now
auto p2 = point(7, 2); //
assert(distance(p1, p2) == 5); // same function works!
```

PRETTY COOL? WAIT!

```
template <typename T, T n>
struct integral_constant {
  // ...

  template <typename F>
  void times(F f) const {
    f(); f(); ... f(); // n times
  }
};
```

LOOP UNROLLING: THEN

```
__attribute__((noinline)) void f() { }

int main() {
    #pragma PLZ UNROLL
    for (int i = 0; i != 10; ++i)
        f();
}
```

LOOP UNROLLING: NOW

```
__attribute__((noinline)) void f() { }

int main() {
    int_<10>.times(f);
}
```

SCEPTICAL?

```
_main:
  ; snip
  push    rbp
  mov rbp, rsp
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  call __Z1fv
  xor eax, eax
  pop rbp
  ret
```

SO FAR SO GOOD

WHAT IF?

```
template <typename ...T>
struct tuple {
  // ...

  template <typename N>
  constexpr decltype(auto) operator[](N const&) {
    return std::get<N::value>(*this);
  }
};
```

ACCESSING AN ELEMENT OF A TUPLE: THEN

```
tuple<int, char, float> values = {1, 'x', 3.4f};
char x = std::get<1>(values);
```

ACCESSING AN ELEMENT OF A TUPLE: NOW

```
tuple<int, char, float> values = {1, 'x', 3.4f};
char x = values[1_c];
```

WHY STOP HERE?

- `std::ratio`
- `std::integer_sequence`

WHAT ARE TYPES?

AREN'T THEY VALUES AFTER ALL?

```
using result = add_pointer<int>::type;
```

LET'S TREAT THEM AS SUCH!

```
class Type {
  // ...
};

Type t{...};
```

WHAT ARE METAFUNCTIONS?

AREN'T THEY FUNCTIONS AFTER ALL?

```
template <typename T>
struct add_pointer {
  using type = T*;
};
```

LET'S TREAT THEM AS SUCH!

```
Type add_pointer(Type);
bool is_pointer(Type);
bool operator==(Type t, Type u);
// ...

Type t{...};
Type p = add_pointer(t);
assert(is_pointer(p));
```

COOL IDEA, BUT USELESS FOR METAPROGRAMMING

EASY FIX: USE TEMPLATES

```
template <typename T>
class Type { /* nothing */ };

Type<int> t{};
```

EASY FIX: USE TEMPLATES

```
template <typename T>
constexpr Type<T*> add_pointer(Type<T> const&)
{ return {}; }

template <typename T>
constexpr std::false_type is_pointer(Type<T> const&)
{ return {}; }

template <typename T>
constexpr std::true_type is_pointer(Type<T*> const&)
{ return {}; }
```

TADAM!

```
Type<int> t{};
auto p = add_pointer(t);
static_assert(is_pointer(p), "");
```

SUGAR

```
template <typename T>
constexpr Type<T> type{};

auto t = type<int>;
auto p = add_pointer(t);
static_assert(is_pointer(p), "");
```

BUT WHAT DOES THAT BUY US?

TYPES ARE NOW FIRST CLASS CITIZENS!

```
auto xs = make_tuple(type<int>, type<char>, type<void>);
auto c = xs[1_c];

// sugar:
auto ys = tuple_t<int, char, void>;
```

FULL LANGUAGE CAN BE USED

Before

```
using ts = vector<int, char&, void*>;
using us = copy_if<ts, or_<std::is_pointer<_1>,
                           std::is_reference<_1>>>::type;
```

After

```
auto ts = make_tuple(type<int>, type<char&>, type<void*>);
auto us = filter(ts, [](auto t) {
  return or_(is_pointer(t), is_reference(t));
});
```

ONLY ONE LIBRARY IS REQUIRED

Before

```
// types (MPL)
using ts = mpl::vector<int, char&, void*>;
using us = mpl::copy_if<ts, mpl::or_<std::is_pointer<_1>,
                                     std::is_reference<_1>>>::type;

// values (Fusion)
auto vs = fusion::make_vector(1, 'c', nullptr, 3.5);
auto ws = fusion::filter_if<std::is_integral<mpl::_1>>(vs);
```

After

```
// types
auto ts = tuple_t<int, char&, void*>;
auto us = filter(ts, [](auto t) {
  return or_(is_pointer(t), is_reference(t));
});

// values
auto vs = make_tuple(1, 'c', nullptr, 3.5);
auto ws = filter(vs, [](auto t) {
  return is_integral(t);
});
```

UNIFIED SYNTAX MEANS MORE REUSE

(AMPHIBIOUS EDSL USING BOOST.PROTO)

```
auto expr = (_1 - _2) / _2;

// compile-time computations
static_assert(decltype(evaluate(expr, 6_c, 2_c))::value == 2, "");

// runtime computations
int i = 6, j = 2;
assert(evaluate(expr, i, j) == 2);
```

UNIFIED SYNTAX MEANS MORE CONSISTENCY

Before

```
auto map = make_map<char, int, long, float, double, void>(
  "char", "int", "long", "float", "double", "void"
);

std::string i = at_key<int>(map);
assert(i == "int");
```

After

```
auto map = make_map(
  make_pair(type<char>,   "char"),
  make_pair(type<int>,    "int"),
  make_pair(type<long>,   "long"),
  make_pair(type<float>,  "float"),
  make_pair(type<double>, "double")
);

std::string i = map[type<int>];
assert(i == "int");
```

SO WE CAN REPRESENT TYPES AS VALUES

BUT HOW CAN WE GET BACK THE TYPES?

```
auto t = add_pointer(type<int>); // could be a complex type computation
using T = the-type-represented-by-t;
// do something useful with T here...
```

EASY FIX: USE NESTED ALIAS

```
template <typename T>
struct Type {
  using type = T;
};
```

AND THEN `decltype`

```
auto t = add_pointer(type<int>);
using T = decltype(t)::type;
static_assert(std::is_same<T, int*>{}, "");
// do something useful with T here...
```

3 STEP PROCESS

1. Wrap types into `type<...>`
2. Perform computations
3. Unwrap with `decltype(...)::type`

ISN'T THAT CUMBERSOME?

NOT REALLY

ONLY DONE AT SOME THIN BOUNDARIES

```
auto t = type<T>;
auto result = huge_type_computation(t);
using Result = decltype(result)::type;
```

NOT ALWAYS REQUIRED

```
auto t = type<T>;
auto result = type_computation(t);
static_assert(result == type<Something>, "");
```

EVEN THEN, IT'S NOT THAT BAD

FINDING SMALLEST TYPE: THEN

```
template <typename ...T>
struct smallest
  : deref<
    typename min_element<
      vector<T...>, less<sizeof_<_1>, sizeof_<_2>>
    >::type
  >
{ };

template <typename ...T>
using smallest_t = typename smallest<T...>::type;

static_assert(std::is_same<
  smallest_t<char, long, long double>, char
>::value, "");
```

FINDING SMALLEST TYPE: NOW

```
template <typename ...T>
auto smallest = minimum(tuple_t<T...>, [](auto t, auto u) {
  return sizeof_(t) < sizeof_(u);
});

template <typename ...T>
using smallest_t = typename decltype(smallest<T...>)::type;

static_assert(std::is_same<
  smallest_t<char, long, long double>, char
>::value, "");
```

SO TYPES ARE OBJECTS

WHAT SHOULD BE THEIR INTERFACE?

IT'S PRECISELY `<type_traits>`!

```
add_pointer  -> std::add_pointer
add_const    -> std::add_const
add_volatile -> std::add_volatile
// ...

is_pointer   -> std::is_pointer
is_const     -> std::is_const
is_volatile  -> std::is_volatile
// ...

operator==  -> std::is_same
// ...
```

THERE'S A GENERIC LIFTING PROCESS

LET'S RECONSIDER:

```
template <typename T>
Type<T*> add_pointer(Type<T>) { return {}; }
```

THIS COULD ALSO BE WRITTEN AS

```
template <typename T>
auto add_pointer(Type<T>) {
  return type<typename std::add_pointer<T>::type>;
}
```

THERE'S A PATTERN

```
template <template <typename ...> class F>
struct Metafunction {
  template <typename ...T>
  auto operator()(Type<T> ...) const {
    return type<
      typename F<T...>::type
    >;
  }
};

Metafunction<std::add_pointer> add_pointer{};

auto i = type<int>;
auto i_ptr = add_pointer(i);
static_assert(is_pointer(i_ptr), "");
```

SUGAR, AGAIN

```
template <template <typename ...> class F>
constexpr Metafunction<F> metafunction{};

auto add_pointer = metafunction<std::add_pointer>;
auto remove_reference = metafunction<std::remove_reference>;
auto add_const = metafunction<std::add_const>;
// ...
```

CASE STUDY: SWITCH FOR `boost::any`

```
boost::any a = 3;
std::string result = switch_<std::string>(a)(
    case_<int>([](int i) { return std::to_string(i); })
  , case_<double>([](double d) { return std::to_string(d); })
  , empty([] { return "empty"; })
  , default_([] { return "default"; })
);
assert(result == "3");
```

DISCLAIMER:

NOT AS GOOD AS SEBASTIAN'S

FIRST

```
template <typename T>
auto case_ = [](auto f) {
  return std::make_pair(hana::type<T>, f);
};

struct _default;
auto default_ = case_<_default>;
auto empty = case_<void>;
```

THE BEAST

```
template <typename Result = void, typename Any>
auto switch_(Any& a) {
  return [&a](auto ...cases_) -> Result {
    auto cases = hana::make_tuple(cases_...);

    auto default_ = hana::find_if(cases, [](auto const& c) {
      return c.first == hana::type<_default>;
    });
    static_assert(!hana::is_nothing(default_),
      "switch is missing a default_ case");

    auto rest = hana::filter(cases, [](auto const& c) {
      return c.first != hana::type<_default>;
    });

    return hana::unpack(rest, [&](auto& ...rest) {
      return impl<Result>(a, a.type(), default_->second, rest...);
    });
  };
}
```

PROCESSING CASES

```
template <typename Result, typename Any, typename Default,
          typename Case, typename ...Rest>
Result impl(Any& a, std::type_index const& t, Default& default_,
            Case& case_, Rest& ...rest)
{
  using T = typename decltype(case_.first)::type;
  if (t == typeid(T)) {
    return hana::if_(hana::type<T> == hana::type<void>,
      [](auto& case_, auto& a) {
        return case_.second();
      },
      [](auto& case_, auto& a) {
        return case_.second(*boost::unsafe_any_cast<T>(&a));
      }
    )(case_, a);
  }
  else
    return impl<Result>(a, t, default_, rest...);
}
```

BASE CASE

```
template <typename Result, typename Any, typename Default>
Result impl(Any&, std::type_index const& t, Default& default_) {
  return default_();
}
```

ABOUT 70 LOC

AND YOUR COWORKERS COULD UNDERSTAND

(MOSTLY)

MY PROPOSAL: HANA

- Heterogeneous + type level computations
- 75+ algorithms
- 8 heterogeneous containers
- Improved compile-times

WORKING ON C++14 COMPILERS

- Clang >= 3.5
- GCC 5 (almost)

AVAILABLE IN YOUR NEXT BOOST RELEASE!

(HOPEFULLY)

FORMAL REVIEW: JUNE 10 - JUNE 24

EMBRACE THE FUTURE

EMBRACE HANA

THANK YOU

http://ldionne.com
http://github.com/ldionne

BONUS

SORTING AT COMPILE-TIME: THEN

```
template <typename Sequence, typename Pred>
struct sort_impl;

template <typename Sequence, typename Pred>
struct sort
  : eval_if<empty<Sequence>,
      identity<Sequence>,
      sort_impl<Sequence, Pred>
  >
{ };
```

SORTING AT COMPILE-TIME: THEN

```
template <typename Sequence, typename Pred>
struct sort_impl {
  using pivot = typename begin<Sequence>::type;
  using parts = typename partition<
    iterator_range<typename next<pivot>::type,
                   typename end<Sequence>::type>
    , apply2<typename lambda<Pred>::type, _1,
             typename deref<pivot>::type>
    , back_inserter<vector<>>
    , back_inserter<vector<>>
  >::type;

  using part1 = typename push_back<
    typename sort<typename parts::first, Pred>::type,
    typename deref<pivot>::type
  >::type;

  using part2 = typename sort<typename parts::second, Pred>::type;

  using type = typename insert_range<
    part1, typename end<part1>::type, part2
  >::type;
};
```

SORTING AT COMPILE-TIME: NOW

```
template <typename Xs, typename Pred>
auto sort(Xs xs, Pred pred) {
  return eval_if(length(xs) < size_t<2>,
    lazy(xs),
    lazy([=](auto xs) {
      auto pivot = head(xs);
      auto parts = partition(tail(xs), partial(pred, pivot));
      return concat(
        append(sort(second(parts), pred), pivot),
        sort(first(parts), pivot)
      );
    })(xs)
  );
}
```

CASE STUDY: INDEXED SORTING

```
auto types = tuple_t<int[3], int[2], int[1]>;
auto indexed = indexed_sort(types, [](auto t, auto u) {
    return sizeof_(t) < sizeof_(u);
});

auto sorted = first(indexed);
auto indices = second(indexed);

static_assert(sorted == tuple_t<int[1], int[2], int[3]>, "");
static_assert(indices == tuple_c<std::size_t, 2, 1, 0>, "");
```

THE BEAST

```
auto indexed_sort = [](auto list, auto predicate) {
  auto indices = to<Tuple>(range(0_c, size(list)));
  auto indexed_list = zip.with(make_pair, list, indices);
  auto sorted = sort(indexed_list, [&](auto const& x, auto const& y) {
    return predicate(first(x), first(y));
  });
  return make_pair(transform(sorted, first), transform(sorted, second));
};
```

WITH FUSION/MPL ...

```
template <typename Types, typename Predicate = quote2<less>>
struct indexed_sort {
  using indexed_types = typename copy<
    zip_view<vector<Types, range_c<long, 0l, size<Types>::value>>>,
    back_inserter<vector<>>
  >::type;

  using sorted = typename sort<
    indexed_types,
    apply2<typename lambda<Predicate>::type, front<_1>, front<_2>>
  >::type;

  using type = pair<
    typename transform<sorted, front<_1>>::type,
    typename transform<sorted, back<_1>>::type
  >;
};
```

FAKING THE INITIAL SEQUENCE: NOW

```
using Sequence = decltype(unpack(sorted, template_<_tuple>))::type;
auto index_map = second(indexed_sort(indices, less));

Sequence s;
int (&a)[3] = s[index_map[0_c]];
int (&b)[2] = s[index_map[1_c]];
int (&c)[1] = s[index_map[2_c]];
```

FAKING THE INITIAL SEQUENCE: THEN

```
using Sequence = fusion::result_of::as_vector<Sorted>::type;
using IndexMap = indexed_sort<Indices>::type::second;

Sequence s;
int (&a)[3] = fusion::at_c<at_c<IndexMap, 0>::type::value>(s);
int (&b)[2] = fusion::at_c<at_c<IndexMap, 1>::type::value>(s);
int (&c)[1] = fusion::at_c<at_c<IndexMap, 2>::type::value>(s);
```

std::common_type: N3843

```
template <typename T, typename U>
using builtin_common_t = std::decay_t<decltype(
  true ? std::declval<T>() : std::declval<U>()
)>;

template <typename, typename ...>
struct ct { };

template <typename T>
struct ct<void, T> : std::decay<T> { };

template <typename T, typename U, typename ...V>
struct ct<void_t<builtin_common_t<T, U>>, T, U, V...>
  : ct<void, builtin_common_t<T, U>, V...>
{ };

template <typename ...T>
struct common_type : ct<void, T...> { };
```

std::common_type: NOW

```
template <typename ...>
auto common_type_impl = nothing;

template <typename T1, typename ...Tn>
auto common_type_impl<T1, Tn...> = monadic_fold<Maybe>(
  tuple_t<Tn...>, type<std::decay_t<T1>>,
  sfinae([](auto t, auto u) -> decltype(
    traits::decay(true ? traits::declval(t) : traits::declval(u))
  ) { return {}; })
);

template <typename ...T>
struct common_type : decltype(common_type_impl<T...>) { };
```