C++ METAPROGRAMMING

A PARADIGM SHIFT

Louis Dionne, CppCon 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

HETEROGENEOUS SEQUENCES

```
std::tuple<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

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

IF ONLY IT WAS THAT EASY

CLAIM 1

WE NEED ALGORITHMS ON HETEROGENEOUS SEQUENCES

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;
```

CLAIM 2

MPL IS REDUNDANT

BUT MOST IMPORTANTLY

C++14 CHANGES EVERYTHING

SEE FOR YOURSELF

CHECKING FOR A MEMBER: THEN

```cpp
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"));
}
```

INTROSPECTION: TOMORROW

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

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

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 HETEROGENEOUS SEQUENCES

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

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

HANDLE USER-DEFINED TYPES

```
template <typename T>
  std::enable_if_t<Struct<T>::value,
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;
```

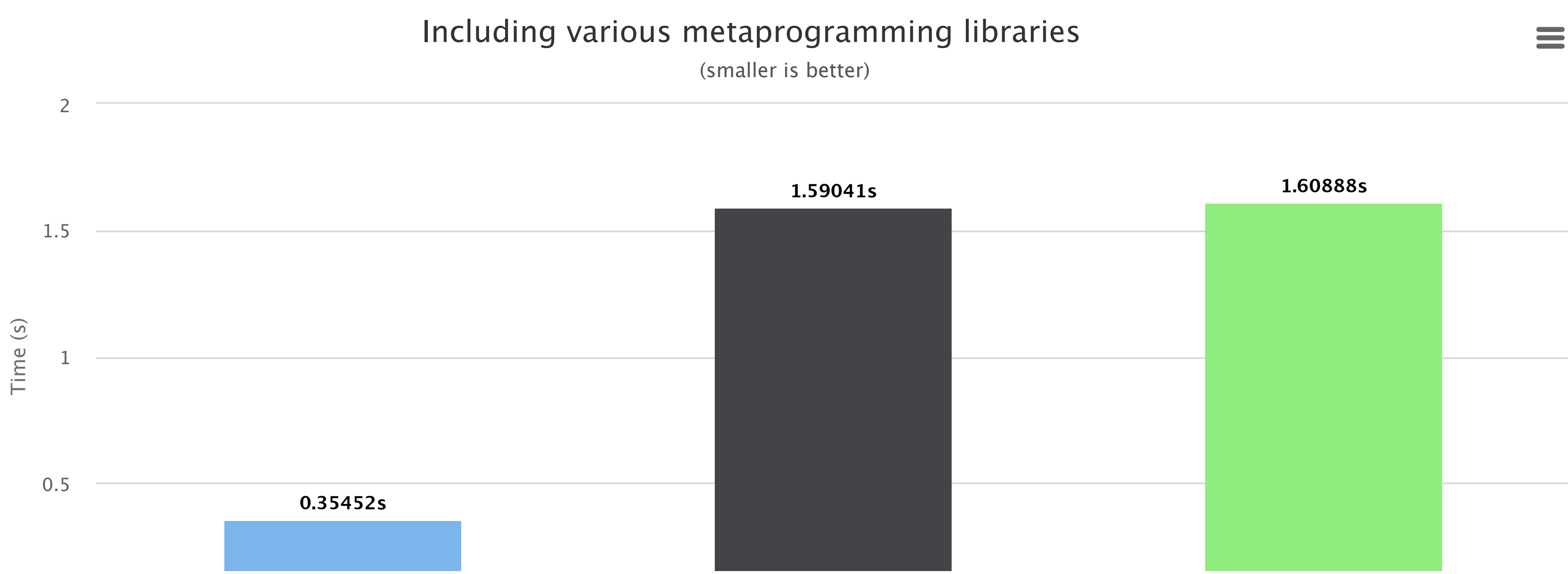
```
boost/mpl/clear.hpp:29:7: error:
  implicit instantiation of undefined template
  'boost::mpl::clear_impl<mpl_::integral_c_tag>::apply<mpl_::int_<1> >'
    : clear_impl< typename sequence_tag<Sequence>::type >
      ^
```

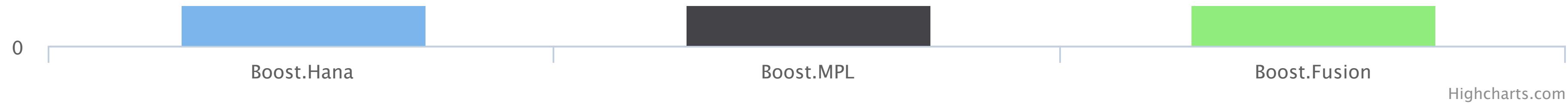
ERROR MESSAGES: NOW

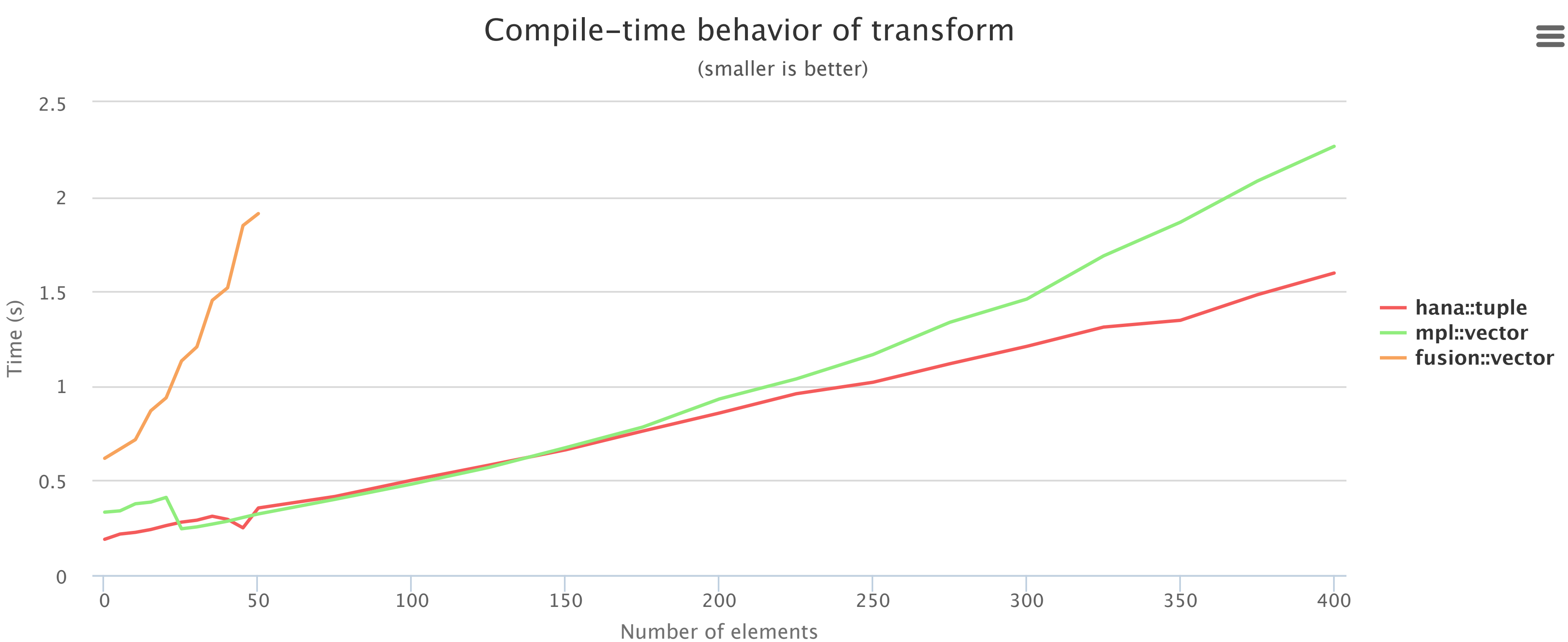
```
auto xs = hana::reverse(1);
```

```
boost/hana/reverse.hpp:35:9:
  static_assert failed
  "hana::reverse(xs) requires 'xs' to be a Sequence"
        static_assert(Sequence<S>::value,
        ^             ~~~~~~~~~~~~~~~~~~
error_messages.cpp:19:24:
  in instantiation of function template specialization
  'boost::hana::reverse_t::operator()<int>' requested here
auto xs = hana::reverse(1);
                       ^
```

COMPILE-TIMES: THEN AND NOW




0
Boost.Hana
Boost.MPL
Boost.Fusion
Highcharts.com


Compile-time behavior of transform
(smaller is better)
Time (s)
Number of elements
hana::tuple
mpl::vector
fusion::vector

WE MUST RETHINK METAPROGRAMMING

BUT HOW?

HERE'S MY TAKE

HEARD OF `integral_constant`?

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

COMPILE-TIME ARITHMETIC: CLASSIC APPROACH

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

static_assert(std::is_same<
  plus<integral_constant<int, 1>, integral_constant<int, 4>>::type,
  integral_constant<int, 5>
>::value, "");
```

THAT'S OK, 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_c{};

static_assert(int_c<1> + int_c<4> == int_c<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

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

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!
```

THAT'S NOT ALL

LOOP UNROLLING

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

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

SCEPTICAL?

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

int main() {
    int_c<5>.times(f);
}
```

Assembly with -O3

```
_main:
  ; snip
  pushq %rbp
  movq  %rsp, %rbp
  callq __Z1fv
  callq __Z1fv
  callq __Z1fv
  callq __Z1fv
  callq __Z1fv
  xorl  %eax, %eax
  popq  %rbp
  retq
```

TUPLE ACCESS

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

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

COMPARE

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

char a = std::get<1>(values);

char b = values[1_c];
```

WHY STOP HERE?

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

HEARD OF <type_traits>?

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

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

LET'S TRY SOMETHING

```
template <typename T>
struct type { };

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_c{};

auto t = type_c<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_c<int>, type_c<char>, type_c<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_c<int>, type_c<char&>, type_c<void*>);
auto us = filter(ts, [](auto t) {
  return 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 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_c<char>,   "char"),
  make_pair(type_c<int>,    "int"),
  make_pair(type_c<long>,   "long"),
  make_pair(type_c<float>,  "float"),
  make_pair(type_c<double>, "double")
);

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

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");
```

FIRST

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

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

THE BEAST

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

    auto default_ = hana::find_if(cases, [](auto const& c) {
      return c.first == hana::type_c<default_t>;
    });
    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_c<default_t>;
    });

    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_c<T> == hana::type_c<void>,
      [](auto& c, auto& a) {
        return c.second();
      },
      [](auto& c, auto& a) {
        return c.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, YOUR DELIVERANCE: HANA

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

WORKING ON C++14 COMPILERS

- Clang >= 3.5
- GCC 5 on the way

NEWLY ACCEPTED IN BOOST!

SOON PART OF THE DISTRIBUTION

EMBRACE THE FUTURE

EMBRACE HANA

THANK YOU

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

BONUS: IMPLEMENTING HETEROGENEOUS ALGORITHMS

transform

```
template <typename ...T, typename F, std::size_t ...i>
auto transform_impl(std::tuple<T...> const& tuple, F const& f,
                    std::index_sequence<i...>)
{
  return std::make_tuple(f(std::get<i>(tuple))...);
}

template <typename ...T, typename F>
auto transform(std::tuple<T...> const& tuple, F const& f) {
  return transform_impl(tuple, f,
            std::make_index_sequence<sizeof...(T)>{});
}
```

EASY SO FAR

BRACE YOURSELVES

HERE COMES `filter`

```
template <typename Tuple, typename Predicate, std::size_t ...i>
auto filter_impl(Tuple const& tuple, Predicate predicate,
                 std::index_sequence<i...>)
{
  using Indices = filter_indices<
    decltype(predicate(std::get<i>(tuple)))::value...
  >;
  return slice(tuple, Indices{});
}

template <typename ...T, typename Predicate>
auto filter(std::tuple<T...> const& tuple, Predicate predicate) {
  return filter_impl(tuple, predicate,
                     std::make_index_sequence<sizeof...(T)>{});
}
```

```cpp
template <typename Tuple, std::size_t ...i>
auto slice(Tuple const& tuple, std::index_sequence<i...>) {
  return std::make_tuple(std::get<i>(tuple)...);
}
```

```
template <bool ...results>
struct filter_indices_helper {
  // ...
};

template <bool ...results>
using filter_indices = typename filter_indices_helper<results...>::
                       template as_index_sequence<>;
```

```cpp
static constexpr bool results_array[sizeof...(results)] = {results...};
static constexpr std::size_t N =
  count(results_array, results_array + sizeof...(results), true);

static constexpr array<std::size_t, N> compute_indices() {
  array<std::size_t, N> indices{};
  std::size_t* out = &indices[0];
  for (std::size_t i = 0; i < sizeof...(results); ++i)
    if (results_array[i])
      *out++ = i;
  return indices;
}

static constexpr array<std::size_t, N> computed_indices
                                      = compute_indices();
```

```cpp
template <typename Indices = std::make_index_sequence<N>>
struct as_index_sequence;

template <std::size_t ...i>
struct as_index_sequence<std::index_sequence<i...>>
  : std::index_sequence<computed_indices[i]...>
{ };
```