METAPROGRAMMING IN C++14 AND BEYOND

LOUIS DIONNE

A BIT OF HISTORY

IT ALL STARTED WITH TEMPLATES

```
template <typename T>
struct vector { /* ... */ };

int main() {
  vector<int> ints = {1, 2, 3};
  vector<string> strings = {"foo", "bar", "baz"};
}
```

WE SUSPECTED THEY WERE HIDING SOMETHING MORE POWERFUL

IT WASN'T CLEAR UNTIL SOMEONE CAME UP WITH A VERY SPECIAL PROGRAM

MARCH 1994, SAN DIEGO MEETING

ERWIN UNRUH COMES UP WITH THIS:

```
template <int i> struct D { D(void*); operator int(); };

template <int p, int i> struct is_prime {
  enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i -1> :: prim };
};

template < int i > struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i, i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum {prim=1}; };
struct is_prime<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim = 1}; void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif
main () { Prime_print<LAST> a; }
```

(source: http://www.erwin-unruh.de/primorig.html)

IT PRINTS PRIME NUMBERS AT COMPILE-TIME

```
P:\HC\D386_O> hc3 i primes.cpp -DLAST=30

MetaWare High C/C++ Compiler R2.6
(c) Copyright 1987-94, MetaWare Incorporated
E "primes.cpp",L16/C63(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<2>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<3>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<5>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<7>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<11>´ [...]
-- Detected during instantiation of Prime_print<30> [...]
E "primes.cpp",L11/C25(#416):   prim
|    Type `enum{}´ can´t be converted to txpe `D<13>´ [...]

[...]
```

FAST FORWARD TO 2001

ANDREI ALEXANDRESCU PUBLISHES MODERN C++ DESIGN

INTRODUCES THE LOKI LIBRARY, WHICH INCLUDES Typelist

```
template <class T, class U>
struct Typelist {
   typedef T Head;
   typedef U Tail;
};

using Types = LOKI_TYPELIST_4(int, char, float, void);
static_assert(std::is_same<
    Loki::TL::TypeAt<Types, 2>::Result,
    float
>{});
```

SEVERAL ALGORITHMS ON `Typelist` ARE PROVIDED

```
using Types = LOKI_TYPELIST_6(int, char, float, char, void, float);

using NoChar = Loki::TL::EraseAll<Types, char>::Result;
using Uniqued = Loki::TL::NoDuplicates<Types>::Result;
using Reversed = Loki::TL::Reverse<Types>::Result;
// etc...
```

THE NOTION OF COMPILE-TIME ALGORITHMS AND DATA STRUCTURES STARTS TO EMERGE

2004

D. ABRAHAMS AND A. GURTOVOY PUBLISH THE MPL BOOK

The book is actually called
C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond

IT MAKES A THOROUGH TREATMENT OF METAPROGRAMMING THROUGH THE BOOST MPL LIBRARY

THE LIBRARY CONTAINS SEVERAL META DATA STRUCTURES

- `boost::mpl::vector`
- `boost::mpl::list`
- `boost::mpl::map`
- `boost::mpl::set`
- `boost::mpl::string`

IT ALSO PROVIDES SEVERAL GENERIC ALGORITHMS WORKING ON META-ITERATORS, LIKE THE STL

- `boost::mpl::equal`
- `boost::mpl::transform`
- `boost::mpl::remove_if`
- `boost::mpl::sort`
- `boost::mpl::partition`
- etc...

FOR EXAMPLE

```
using Types = mpl::vector<int, void, char, long, void>;

using NoVoid = mpl::remove_if<Types, std::is_void<mpl::_1>>::type;
static_assert(mpl::equal<mpl::vector<int, char, long>,
                         NoVoid>{});

using Ptrs = mpl::transform<Types, std::add_pointer<mpl::_1>>::type;
static_assert(mpl::equal<Ptrs,
                mpl::vector<int*, void*, char*, long*, void*>>{});
```

2008

J. DE GUZMAN, D. MARSDEN AND T. SCHWINGER RELEASE THE BOOST FUSION LIBRARY

MPL ALLOWS MANIPULATING TYPES (AT COMPILE-TIME)

FUSION ALLOWS MANIPULATING OBJECTS (AT COMPILE-TIME)

LIKE MPL, IT PROVIDES DATA STRUCTURES

- `boost::fusion::vector`
- `boost::fusion::list`
- `boost::fusion::set`
- `boost::fusion::map`

AND ALGORITHMS

- `boost::fusion::remove_if`
- `boost::fusion::find_if`
- `boost::fusion::count_if`
- `boost::fusion::transform`
- `boost::fusion::reverse`
- etc...

FOR EXAMPLE

```
// vector
auto vector = fusion::make_vector(1, 2.2f, "hello"s, 3.4, 'x');
auto no_floats = fusion::remove_if<
                      std::is_floating_point<mpl::_>>(vector);

assert(no_floats == fusion::make_vector(1, "hello"s, 'x'));

// map
struct a; struct b; struct c;
auto map = fusion::make_map<a, b, c>(1, 'x', "hello"s);

assert(fusion::at_key<a>(map) == 1);
assert(fusion::at_key<b>(map) == 'x');
assert(fusion::at_key<c>(map) == "hello"s);
```

BOOSTCON 2010

MATT CALABRESE AND ZACH LAINE PRESENT INSTANTIATIONS MUST GO

THEY INTRODUCE A WAY OF METAPROGRAMMING WITHOUT ANGLY BRACKETS

THE IDEA IS KINDA SHOT DOWN AND NOBODY FOLLOWS UP

...UNTIL HANA

WHAT WE CAN DO TODAY

SOME FEATURES MAKE IT EASIER AND FASTER

- Variadic templates
- Alias templates

BRIGAND

```
using Types = brigand::list<int, void, char, long, void>;

using NoVoid = brigand::remove_if<Types, std::is_void<brigand::_1>>;
static_assert(std::is_same<NoVoid,
              brigand::list<int, char, long>>{});

using Ptrs = brigand::transform<Types, std::add_pointer<brigand::_1>>;
static_assert(std::is_same<Ptrs,
              brigand::list<int*, void*, char*, long*, void*>>{});
```

METAL

```
using Types = metal::list<int, void, char, long, void>;

using NoVoid = metal::remove_if<Types, metal::lambda<is_void>>;
static_assert(std::is_same<NoVoid,
              metal::list<int, char, long>>{});

using Ptrs = metal::transform<metal::lambda<std::add_pointer_t>,Types>;
static_assert(std::is_same<Ptrs,
              metal::list<int*, void*, char*, long*, void*>>{});
```

BUT OTHER FEATURES COMPLETELY CHANGE THE PICTURE

- Function return type deduction
- Generic lambdas
- `decltype`

WHAT IS THIS "NEW PICTURE"?

ESSENTIALLY: REPRESENT COMPILE-TIME ENTITIES AS OBJECTS

INTEGERS

```
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>
using plus = integral_constant<
  decltype(X::value + Y::value),
  X::value + Y::value
>;

using result = plus<integral_constant<int, 3>,
                    integral_constant<int, 4>>;
```

BUT WHAT ABOUT THIS?

```
template <typename T, T x, T y>
constexpr auto
operator+(integral_constant<T, x>, integral_constant<T, y>)
{ return integral_constant<T, x + y>{}; }

auto result = integral_constant<int, 3>{} + integral_constant<int, 4>{};
```

NOW, WE CAN ACTUALLY DO MORE

```
template <char ...c>
constexpr auto operator"" _c() {
  constexpr int n = parse<c...>();
  return integral_constant<int, n>{};
}

auto n = 10_c + 30_c; // n is integral_constant<int, 40>
```

MANIPULATING TYPES

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

template <typename T>
constexpr type<T*> add_pointer(type<T>)
{ return {}; }

template <typename T, typename U>
constexpr std::false_type operator==(type<T>, type<U>)
{ return {}; }

template <typename T>
constexpr std::true_type operator==(type<T>, type<T>)
{ return {}; }
```

USUAL FUNCTION CALL SYNTAX WORKS

Before

```
using IntPtr = add_pointer<int>::type;
static_assert(is_same<IntPtr, int*>{});
```

After

```
constexpr auto IntPtr = add_pointer(type<int>{});
static_assert(IntPtr == type<int*>{});
```

WE CAN USE TUPLES INSTEAD OF TYPE LISTS

```
constexpr auto Types = tuple<
  type<int>, type<char>, type<float>, type<char>, type<void>
>{};

// or
template <typename ...T>
constexpr auto tuple_t = tuple<type<T>...>{};

constexpr auto Types = tuple_t<int, char, float, char, void>;
```

PAYS OFF FOR COMPLEX COMPUTATIONS

Before

```
using Types = mpl::vector<int, bool const, char volatile, long const>;
using CV_Types = mpl::copy_if<Types,
                    mpl::or_<std::is_volatile<mpl::_1>,
                             std::is_const<mpl::_1>>>::type;
```

After

```
auto Types = hana::tuple_t<int, bool const, char volatile, long const>;
auto CV_Types = hana::filter(Types, [](auto t) {
  return hana::traits::is_volatile(t) || hana::traits::is_const(t);
});
```

COMPILE-TIME STRINGS

```
constexpr auto Hello_world = "hello"_s + " world"_s;
static_assert(Hello_world == "hello world"_s);
```

HOW THAT WORKS

```
template <char ...c> struct string { };

template <typename CharT, CharT ...c>
constexpr string<c...> operator"" _s() { return {}; }

template <char ...c1, char ...c2>
constexpr auto operator==(string<c1...>, string<c2...>) {
  return std::is_same<string<c1...>, string<c2...>>{};
}

template <char ...c1, char ...c2>
constexpr auto operator+(string<c1...>, string<c2...>) {
  return string<c1..., c2...>{};
}
```

WE CAN USE THIS PRINCIPLE ON ANY COMPILE-TIME "THING"

`std::ratio` comes to mind

ENTER HANA

- data structures like Boost.Fusion
- algorithms like Boost.Fusion
- a way to represent types as values

ALL YOU NEED FROM MPL AND FUSION IN A SINGLE LIBRARY

DATA STRUCTURES

- `boost::hana::tuple`
- `boost::hana::map`
- `boost::hana::set`

ALGORITHMS

- `boost::hana::remove_if`
- `boost::hana::find_if`
- `boost::hana::count_if`
- `boost::hana::transform`
- `boost::hana::reverse`
- etc…

UTILITIES

- `boost::hana::type`
- `boost::hana::integral_constant`
- `boost::hana::string`

FOR EXAMPLE, MPL

```
constexpr auto Types = hana::tuple_t<int, void, char, long>;

constexpr auto NoVoid = hana::remove_if(Types, hana::traits::is_void);
static_assert(NoVoid == hana::tuple_t<int, char, long>);

constexpr auto Ptrs = hana::transform(Types, hana::traits::add_pointer);
static_assert(Ptrs == hana::tuple_t<int*, void*, char*, long*>);
```

FOR EXAMPLE, FUSION

```
// tuple
auto tuple = hana::make_tuple(1, 2.2f, "hello"s, 3.4, 'x');
auto no_floats = hana::remove_if(tuple, [](auto const& t) {
  return hana::traits::is_floating_point(hana::typeid_(t));
});

assert(no_floats == hana::make_tuple(1, "hello"s, 'x'));

// map
struct a; struct b; struct c;
auto map = hana::make_map(
  hana::make_pair(hana::type<a>{}, 1),
  hana::make_pair(hana::type<b>{}, 'x'),
  hana::make_pair(hana::type<c>{}, "hello"s)
);

assert(map[hana::type<a>{}] == 1);
assert(map[hana::type<b>{}] == 'x');
assert(map[hana::type<c>{}] == "hello"s);
```

LET'S CONSIDER A SIMPLE EVENT SYSTEM

```
int main() {
  event_system events{{"foo", "bar", "baz"}};

  events.on("foo", []() { std::cout << "foo triggered!" << '\n'; });
  events.on("foo", []() { std::cout << "foo again!"     << '\n'; });
  events.on("bar", []() { std::cout << "bar triggered!" << '\n'; });
  events.on("baz", []() { std::cout << "baz triggered!" << '\n'; });

  events.trigger("foo");
  events.trigger("baz");
  // events.trigger("unknown"); // WOOPS! Runtime error!
}
```

WHAT IF

- All events are known at compile-time
- We always know what event to trigger at compile-time

COULD WE DO BETTER?

```
int main() {
  auto events = make_event_system("foo"_s, "bar"_s, "baz"_s);

  events.on("foo"_s, []() { std::cout << "foo triggered!" << '\n'; });
  events.on("foo"_s, []() { std::cout << "foo again!" << '\n'; });
  events.on("bar"_s, []() { std::cout << "bar triggered!" << '\n'; });
  events.on("baz"_s, []() { std::cout << "baz triggered!" << '\n'; });
  // events.on("unknown"_s, []() { }); // compiler error!

  events.trigger("foo"_s); // no overhead for event lookup
  events.trigger("baz"_s);
  // events.trigger("unknown"_s); // compiler error!
}
```

RUNTIME

```
struct event_system {
  using Callback = std::function<void()>;
  std::unordered_map<std::string, std::vector<Callback>> map_;
```

COMPILE-TIME

```
template <typename ...Events>
struct event_system {
  using Callback = std::function<void()>;
  hana::map<hana::pair<Events, std::vector<Callback>>...> map_;
```

RUNTIME

```
explicit event_system(std::initializer_list<std::string> events) {
  for (auto const& event : events)
    map_.insert({event, {}});
}
```

COMPILE-TIME

```
template <typename ...Events>
event_system<Events...> make_event_system(Events ...events) {
  return {};
}
```

RUNTIME

```
template <typename F>
void on(std::string const& event, F callback) {
  auto callbacks = map_.find(event);
  assert(callbacks != map_.end() &&
    "trying to add a callback to an unknown event");

  callbacks->second.push_back(callback);
}
```

COMPILE-TIME

```
template <typename Event, typename F>
void on(Event e, F callback) {
  auto is_known_event = hana::contains(map_, e);
  static_assert(is_known_event,
    "trying to add a callback to an unknown event");

  map_[e].push_back(callback);
}
```

RUNTIME

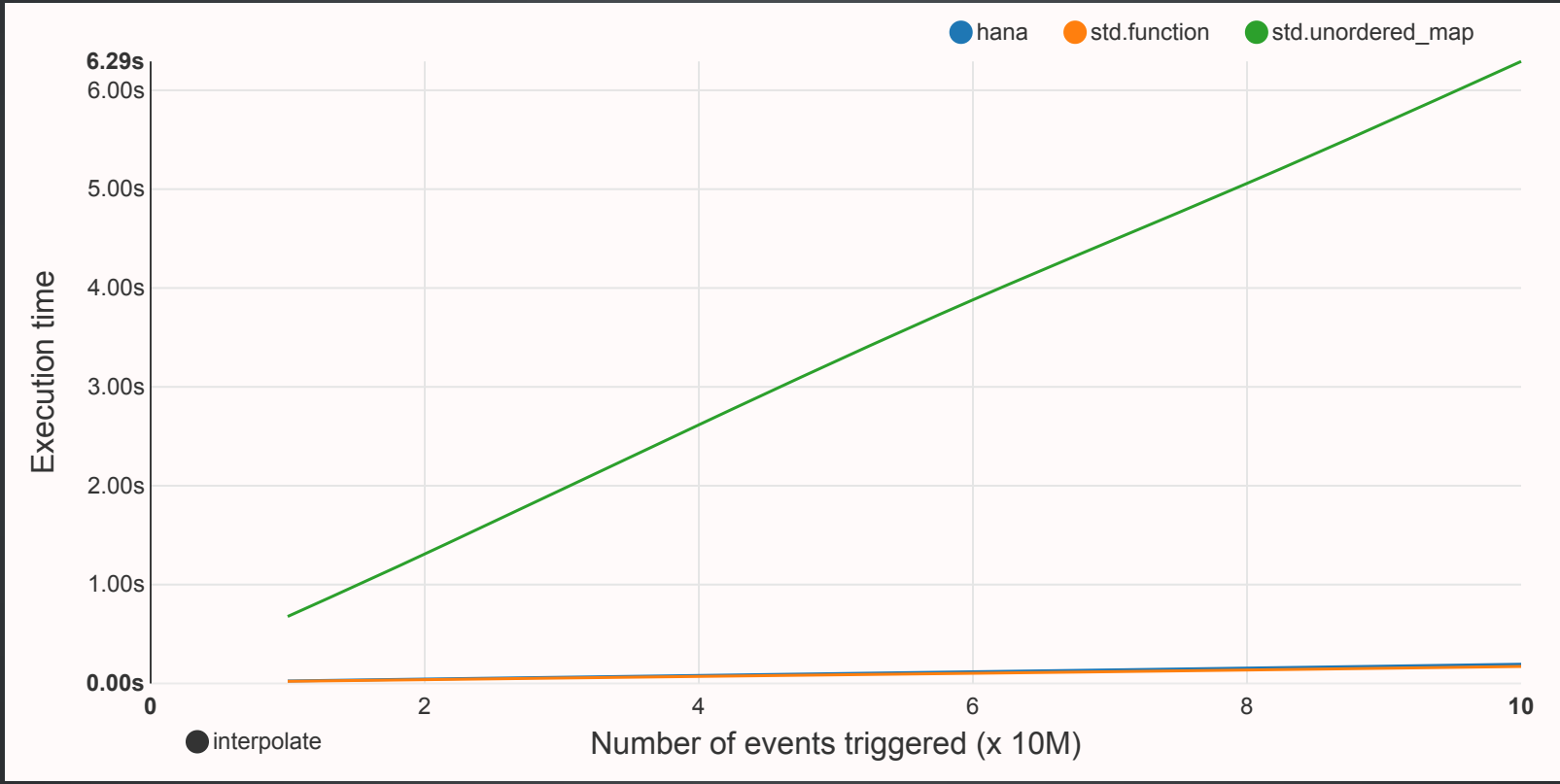
```
void trigger(std::string const& event) const {
  auto callbacks = map_.find(event);
  assert(callbacks != map_.end() &&
    "trying to trigger an unknown event");

  for (auto& callback : callbacks->second)
    callback();
}
```

COMPILE-TIME

```
template <typename Event>
void trigger(Event e) const {
  auto is_known_event = hana::contains(map_, e);
  static_assert(is_known_event,
    "trying to trigger an unknown event");

  for (auto& callback : map_[e])
    callback();
}
```

BUT DOES IT ACTUALLY MATTER?

COMPILED WITH `-O3 -flto`



WHAT IF THE EVENT TO TRIGGER CAN BE DECIDED AT RUNTIME?

```
int main() {
  auto events = make_event_system("foo"_s, "bar"_s, "baz"_s);

  events.on("foo"_s, []() { std::cout << "foo triggered!" << '\n'; });
  events.on("foo"_s, []() { std::cout << "foo again!" << '\n'; });
  events.on("bar"_s, []() { std::cout << "bar triggered!" << '\n'; });
  events.on("baz"_s, []() { std::cout << "baz triggered!" << '\n'; });

  std::string e = read_from_stdin();
  events.trigger(e);
}
```

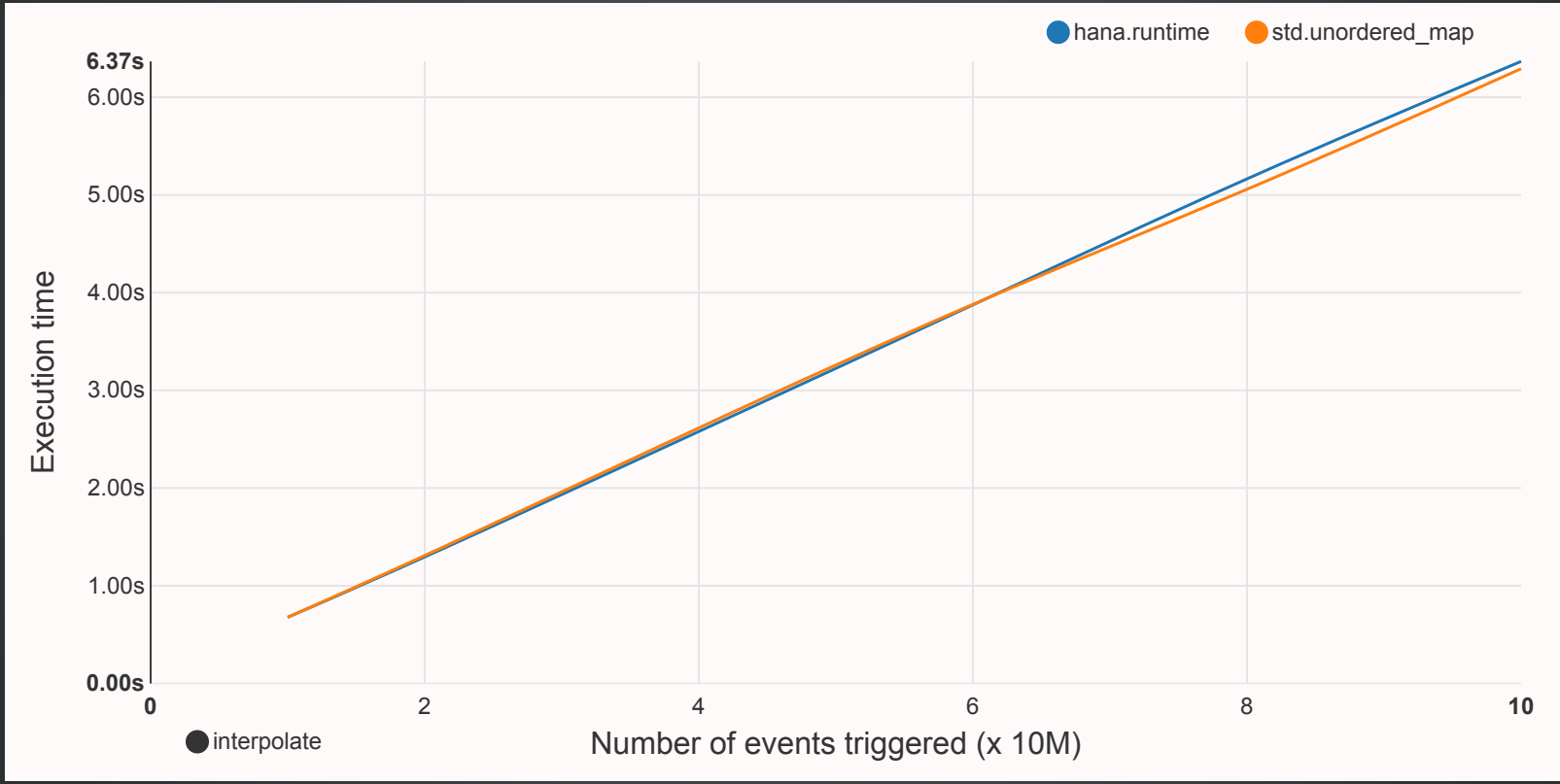
FIRST, MAINTAIN A DYNAMIC MAP

```
std::unordered_map<std::string, std::vector<Callback>* const> dynamic_;

event_system() {
  hana::for_each(hana::keys(map_), [&](auto event) {
    dynamic_.insert({event.c_str(), &map_[event]});
  });
}
```

THEN, OVERLOAD `trigger`!

```
void trigger(std::string const& e) const {
  auto callbacks = dynamic_.find(e);
  assert(callbacks != dynamic_.end() &&
    "trying to trigger an unknown event");

  for (auto& callback : *callbacks->second)
    callback();
}
```

AND WHAT ABOUT PERFORMANCE?



HANA SHINES WHEN COMBINING COMPILE-TIME AND RUNTIME

ANOTHER EXAMPLE

GO INTERFACES

HASKELL TYPECLASSES

RUST TRAITS

C++0X CONCEPT MAPS

NAME THEM HOWEVER YOU'D LIKE

```
struct Square {
  void draw(std::ostream& out) const { out << "Square"; }
};

struct Circle {
  void draw(std::ostream& out) const { out << "Circle"; }
};

void f(drawable const& d) {
  d.draw(std::cout);
}

f(Square{}); // prints "Square"
f(Circle{}); // prints "Circle"
```

```cpp
#include <dyno.hpp>
#include <iostream>
using namespace dyno::literals;

// Define the interface of something that can be drawn
struct Drawable : decltype(dyno::requires(
  "draw"_s = dyno::function<void (dyno::T const&, std::ostream&)>
)) { };

// Define an object that can hold anything that can be drawn.
struct drawable {
  template <typename T>
  drawable(T x) : poly_{x, dyno::make_concept_map(
    "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
  )} { }

  void draw(std::ostream& out) const
  { poly_.virtual_("draw"_s)(poly_, out); }

private:
  dyno::poly<Drawable> poly_;
};
```

WOAH! HOW DOES IT WORK?

IT'S SIMPLE

NO, NOT REALLY

BUT BITS OF IT ARE

HOW WE CREATE THE VTABLE

```
template <typename ...Name, typename ...Signature>
auto requires(hana::pair<Name, hana::type<Signature>> ...f)
  -> hana::map<hana::pair<Name, Signature>...>
;

using VTable = decltype(requires(
  hana::make_pair(
    "draw"_s,
    hana::type<void (void const*, std::ostream&)>{}
  )
));
```

HOW WE FILL IT

```
template <typename ...Name, typename ...Function>
auto make_concept_map(hana::pair<Name, Function> ...f) {
  return hana::make_map(f...);
}

template <typename T>
auto functions = make_concept_map(
  hana::make_pair(
    "draw"_s,
    [](T const& self, std::ostream& out) { self.draw(out); }
  )
);
```

HOW WE BIND THE TWO TOGETHER

```
struct drawable {
  template <typename T>
  drawable(T t) : vtable_{functions<T>}, ... { }

  ...

private:
  VTable vtable_;
  ...
};
```

THEN THROW SOME NICE DSL ON TOP AND YOU GET THIS

```
struct Drawable : decltype(dyno::requires(
  "draw"_s = dyno::function<void (dyno::T const&, std::ostream&)>
)) { };

struct drawable {
  template <typename T>
  drawable(T x) : poly_{x, dyno::make_concept_map(
    "draw"_s = [](T const& self, std::ostream& out) { self.draw(out); }
  )} { }

  void draw(std::ostream& out) const
  { poly_.virtual_("draw"_s)(poly_, out); }

private:
  dyno::poly<Drawable> poly_;
};
```

THE FUTURE

HOW WOULD WE WANT METAPROGRAMMING TO LOOK LIKE?

CONSIDER SERIALIZATION TO JSON

```
struct point { float x, y, z; };
struct triangle { point a, b, c; };

struct tetrahedron {
  triangle base;
  point apex;
};

int main() {
  tetrahedron t{
    {{0.f,0.f,0.f}, {1.f,0.f,0.f}, {0.f,0.f,1.f}},
    {0.f,1.f,0.f}
  };

  to_json(std::cout, t);
}
```

SHOULD OUTPUT

```
{
  "base": {
    "a": {"x": 0, "y": 0, "z": 0},
    "b": {"x": 1, "y": 0, "z": 0},
    "c": {"x": 0, "y": 0, "z": 1}
  },
  "apex": {"x": 0, "y": 1, "z": 0}
}
```

HOW TO WRITE THIS `to_json`?

EASY WITH REFLECTION AND TUPLE FOR-LOOPS

SYNTAX TBD

```
template <typename T>
std::ostream& to_json(std::ostream& out, T const& v) {
  if constexpr (std::meta::Record(reflexpr(T))) {
    out << "{";
    constexpr auto members = reflexpr(T).members();
    for constexpr (int i = 0; i != members.size(); ++i) {
      if (i > 0) out << ", ";
      out << '"' << members[i].name() << "\": ";
      to_json(out, v.*members[i].pointer());
    }
    out << '}';
  } else {
    out << v;
  }
  return out;
}
```

THE FUTURE OF TYPE-LEVEL COMPUTATIONS?

```
constexpr std::vector<std::meta::type>
sort_by_alignment(std::vector<std::meta::type> types) {
  std::sort(v.begin(), v.end(), [](std::meta::type t,
                                   std::meta::type u) {
    return t.alignment() < u.alignment();
  });
  return v;
}

constexpr std::vector<std::meta::type> types{
  reflexpr(Foo), reflexpr(Bar), reflexpr(Baz)
};

constexpr std::vector<std:meta::type> sorted = sort_by_alignment(types);

std::tuple<typename(sorted)...> tuple{...};
```

METAPROGRAMMING IS POWERFUL

WE NEED MORE METAPROGRAMMING

BUT LESS *TEMPLATE* METAPROGRAMMING

LET'S EMBRACE THIS REALITY

https://a9.com/careers