TOWARDS PAINLESS METAPROGRAMMING

LOUIS DIONNE, C++NOW 2014

OUTLINE

- Metaprogramming in C++1y
- Towards a MPL successor: state of affairs

ACT 1
METAPROGRAMMING IN C++1Y

BOILERPLATE FOR THIS SECTION

```
#include <array>
#include <cstddef>
#include <functional>
#include <type_traits>
```

```cpp
template <bool v>
using bool_ = std::integral_constant<bool, v>;

using true_ = bool_<true>;
using false_ = bool_<false>;
```

```
template <typename ...xs>
struct list;

template <typename x, typename xs>
struct cons;

template <typename x, typename ...xs>
struct cons<x, list<xs...>> {
    using type = list<x, xs...>;
};
```

```
template <typename xs>
struct tail;

template <typename x, typename ...xs>
struct tail<list<x, xs...>> {
    using type = list<xs...>;
};
```

```
template <typename xs>
struct head;

template <typename x, typename ...xs>
struct head<list<x, xs...>> {
    using type = x;
};
```

```
template <typename xs>
struct is_empty;

template <typename ...xs>
struct is_empty<list<xs...>>
    : bool_<sizeof...(xs) == 0>
{ };
```

METAFUNCTION MAPPING

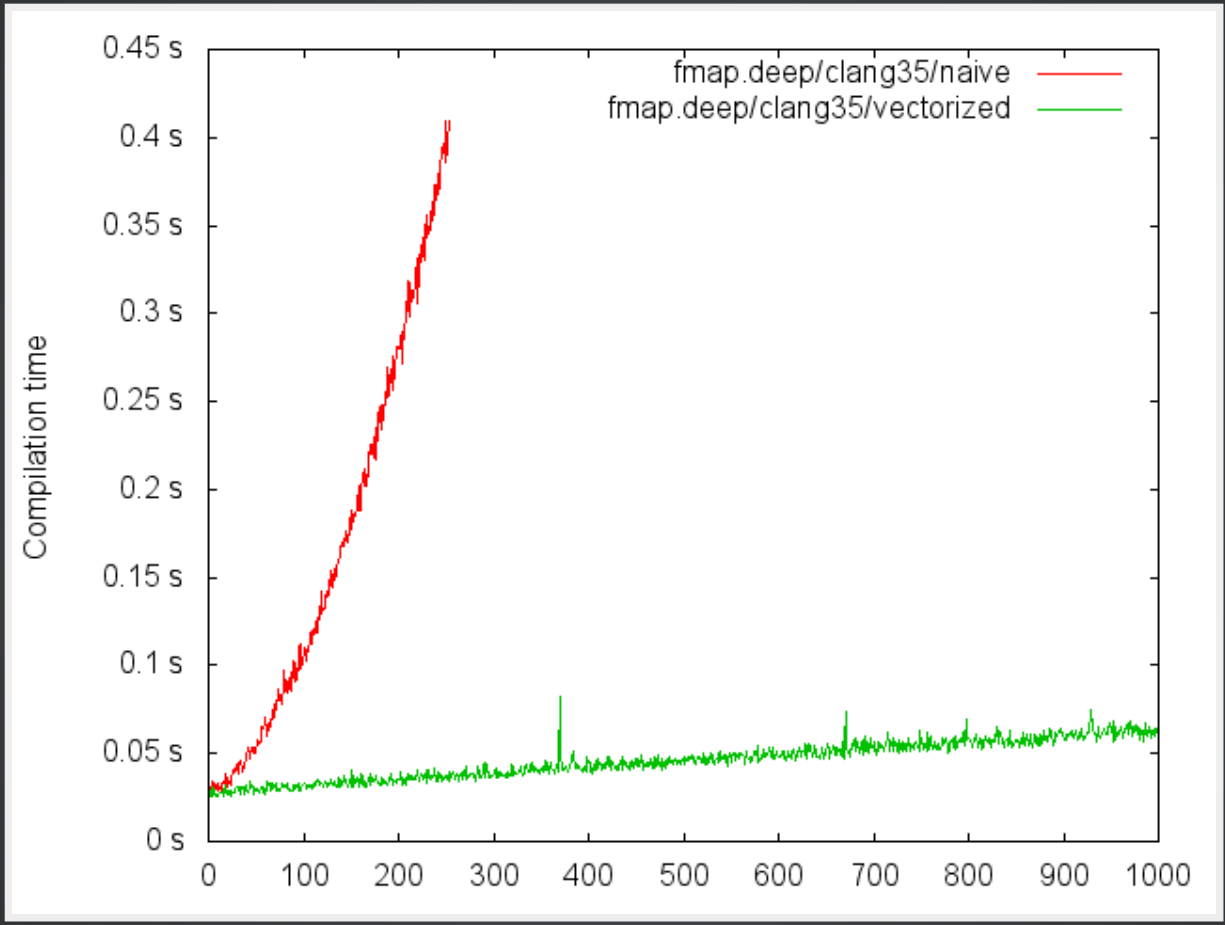
Naive

```
template <template <typename ...> class f, typename xs,
          bool = is_empty<xs>::value>
struct map
    : cons<
        typename f<typename head<xs>::type>::type,
        typename map<f, typename tail<xs>::type>::type
    >
{ };

template <template <typename ...> class f, typename xs>
struct map<f, xs, true> {
    using type = xs;
};
```

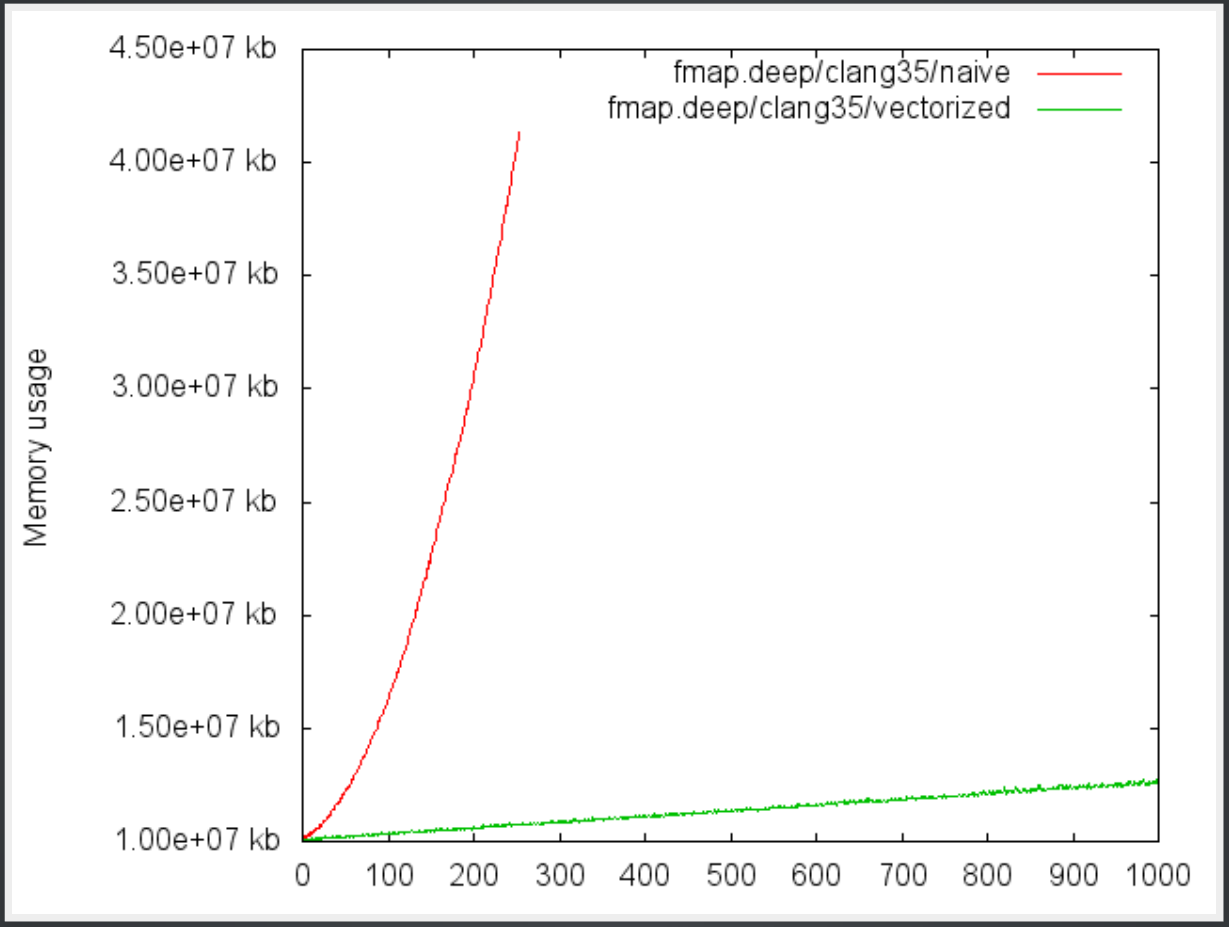
"Vectorized" metafunction application

```
template <template <typename ...> class f, typename xs>
struct map;

template <template <typename ...> class f, typename ...xs>
struct map<f, list<xs...>> {
    using type = list<typename f<xs>::type...>;
};
```

TIME


fmap.deep/clang35/naive
fmap.deep/clang35/vectorized
Compilation time
0.45 s
0.4 s
0.35 s
0.3 s
0.25 s
0.2 s
0.15 s
0.1 s
0.05 s
0 s
0
100
200
300
400
500
600
700
800
900
1000

MEMORY USAGE



LOGICAL OPERATIONS

(without short-circuiting)

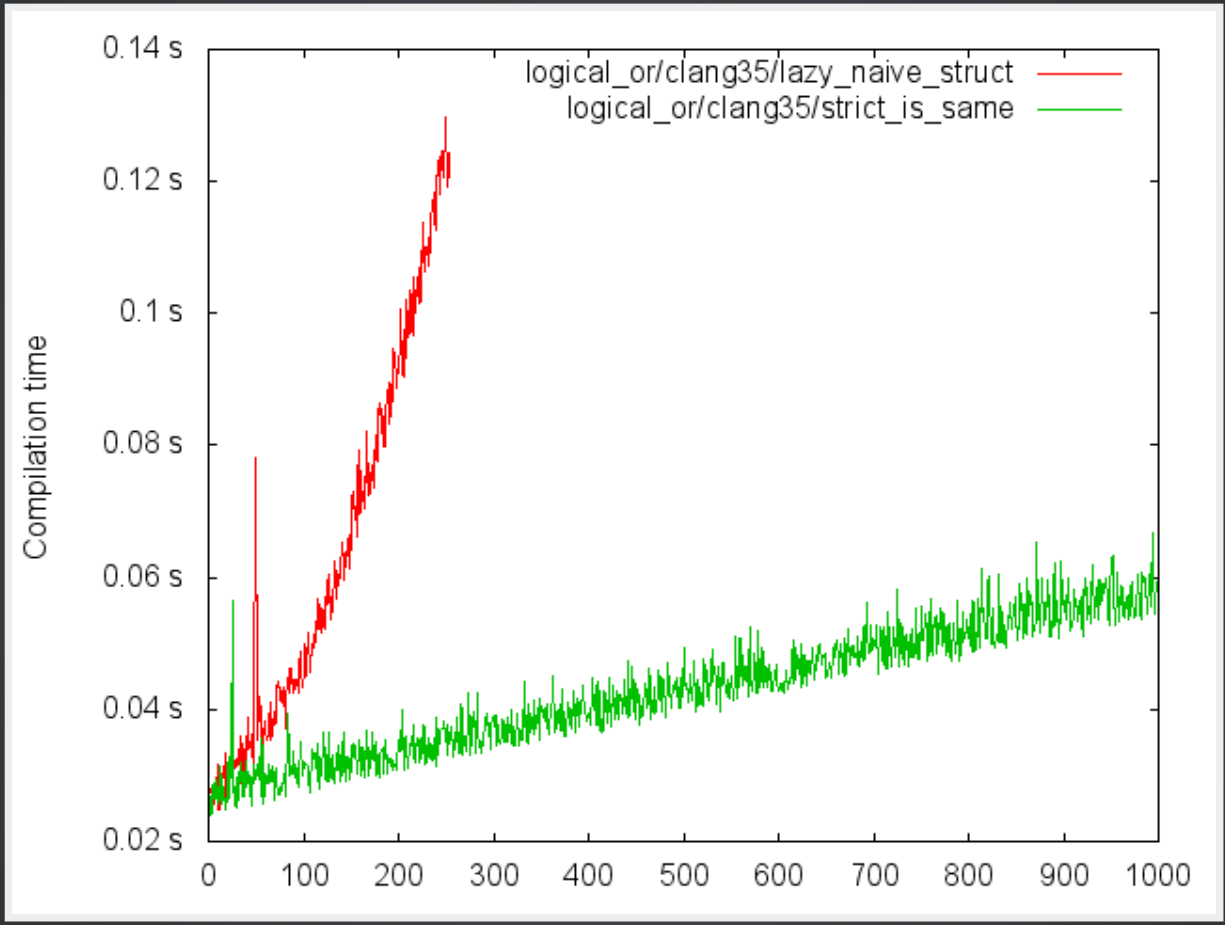
Naive

```
template <typename ...xs>
struct and_ : true_ { };

template <typename x, typename ...xs>
struct and_<x, xs...>
    : std::conditional_t<x::value, and_<xs...>, x>
{ };
```

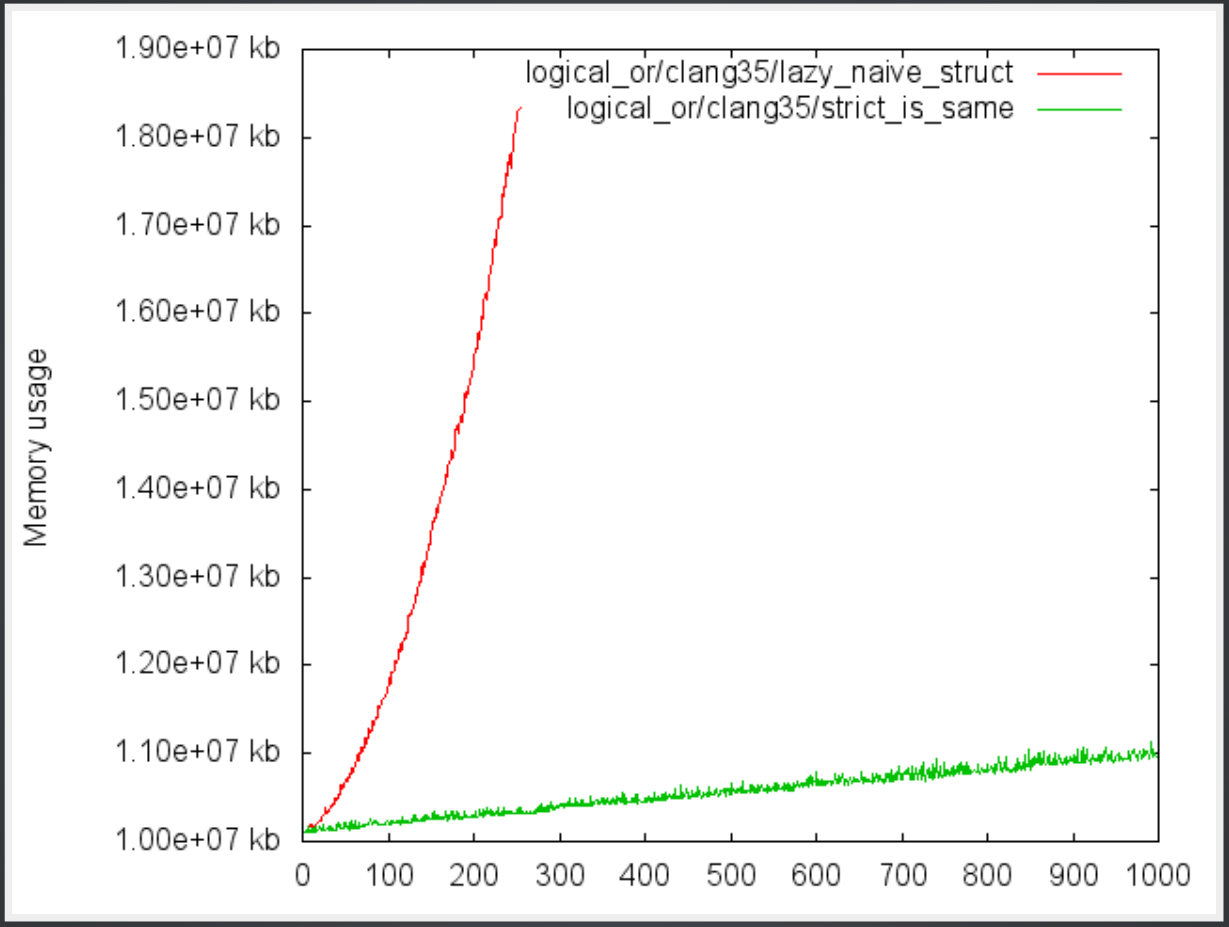
With `std::is_same`

```
template <bool ...> struct bool_seq;

template <typename ...xs>
using and_ = std::is_same<
    bool_seq<xs::value...>,
    bool_seq<(xs::value, true)...>
>;
```

TIME


logical_or/clang35/lazy_naive_struct
logical_or/clang35/strict_is_same
Compilation time
0.14 s
0.12 s
0.1 s
0.08 s
0.06 s
0.04 s
0.02 s
0
100
200
300
400
500
600
700
800
900
1000

MEMORY USAGE



KEY-BASED LOOKUP

In the following slides...

```
template <typename x>
struct no_decay { using type = x; };

template <typename key, typename value>
struct pair { };
```

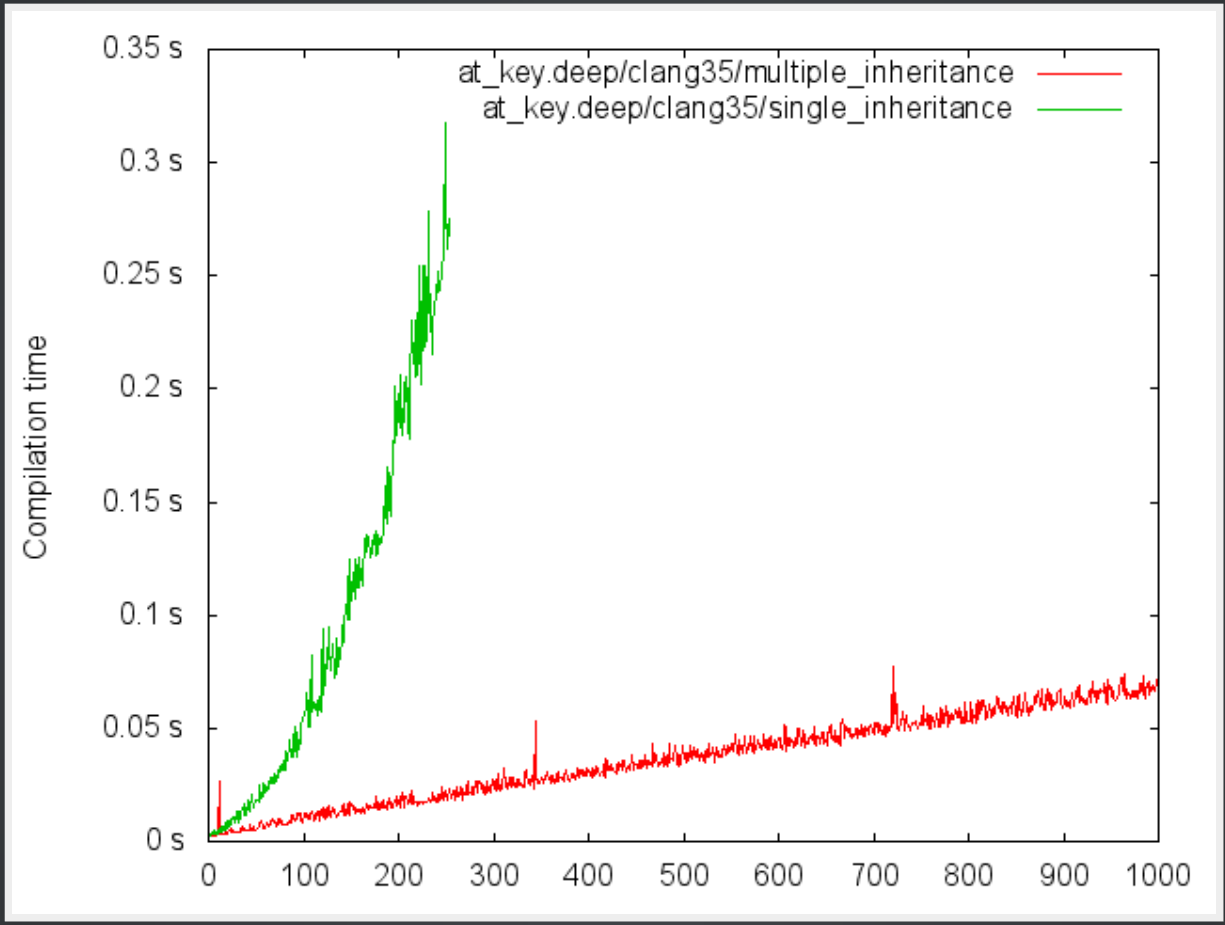
With single inheritance

```
template <typename ...pairs>
struct map {
    template <typename fail = void>
    static typename fail::key_not_found at_key(...);
};

template <typename key, typename value, typename ...pairs>
struct map<pair<key, value>, pairs...> : map<pairs...> {
    using map<pairs...>::at_key;
    static no_decay<value> at_key(no_decay<key>*);
};

template <typename key, typename ...pairs>
using at_key = decltype(
    map<pairs...>::at_key((no_decay<key>*)nullptr)
);
```

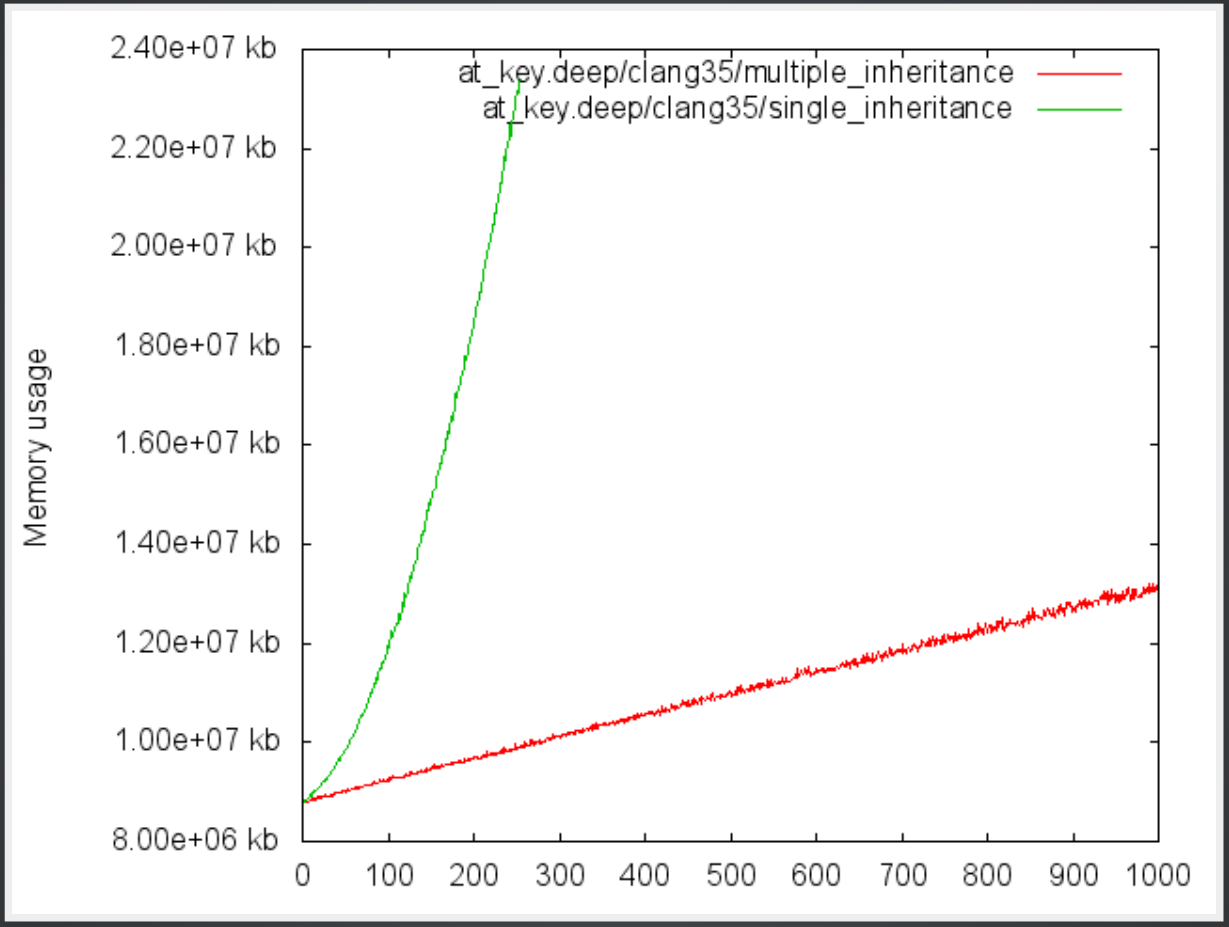
With multiple inheritance

```
template <typename ...xs>
struct inherit : xs... { };

template <typename key, typename value>
static no_decay<value> lookup(pair<key, value>*);

template <typename key, typename ...pairs>
using at_key = decltype(lookup<key>((inherit<pairs...>*)nullptr));
```

TIME


at_key.deep/clang35/multiple_inheritance
at_key.deep/clang35/single_inheritance
Compilation time
0.35 s
0.3 s
0.25 s
0.2 s
0.15 s
0.1 s
0.05 s
0 s
0
100
200
300
400
500
600
700
800
900
1000

MEMORY USAGE



INDEX-BASED LOOKUP

In the following slides...

```
template <typename x>
struct no_decay { using type = x; };

template <std::size_t index, typename value>
struct index_pair { };
```

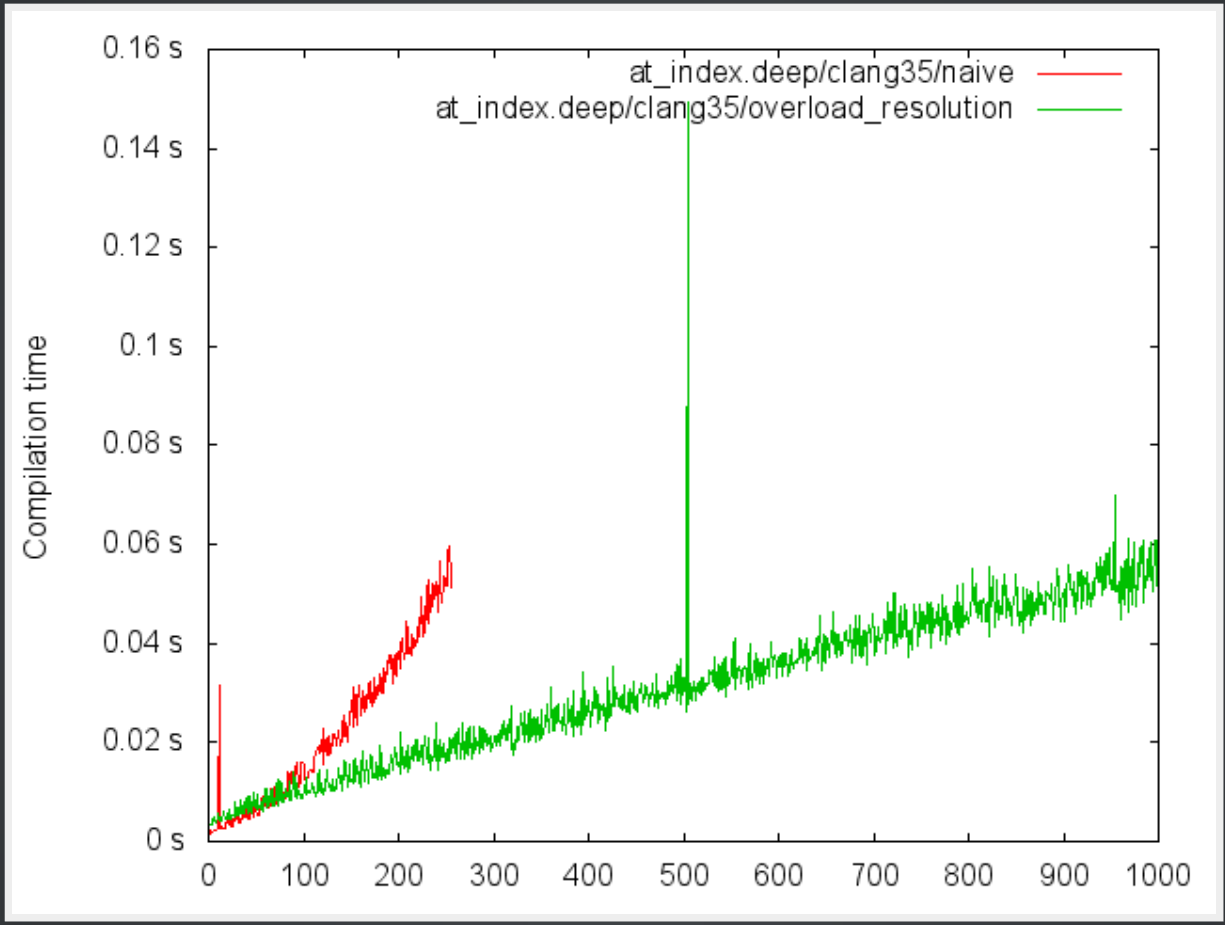
Naive

```
template <std::size_t index, typename x, typename ...xs>
struct at
    : at<index - 1, xs...>
{ };

template <typename x, typename ...xs>
struct at<0, x, xs...> {
    using type = x;
};
```

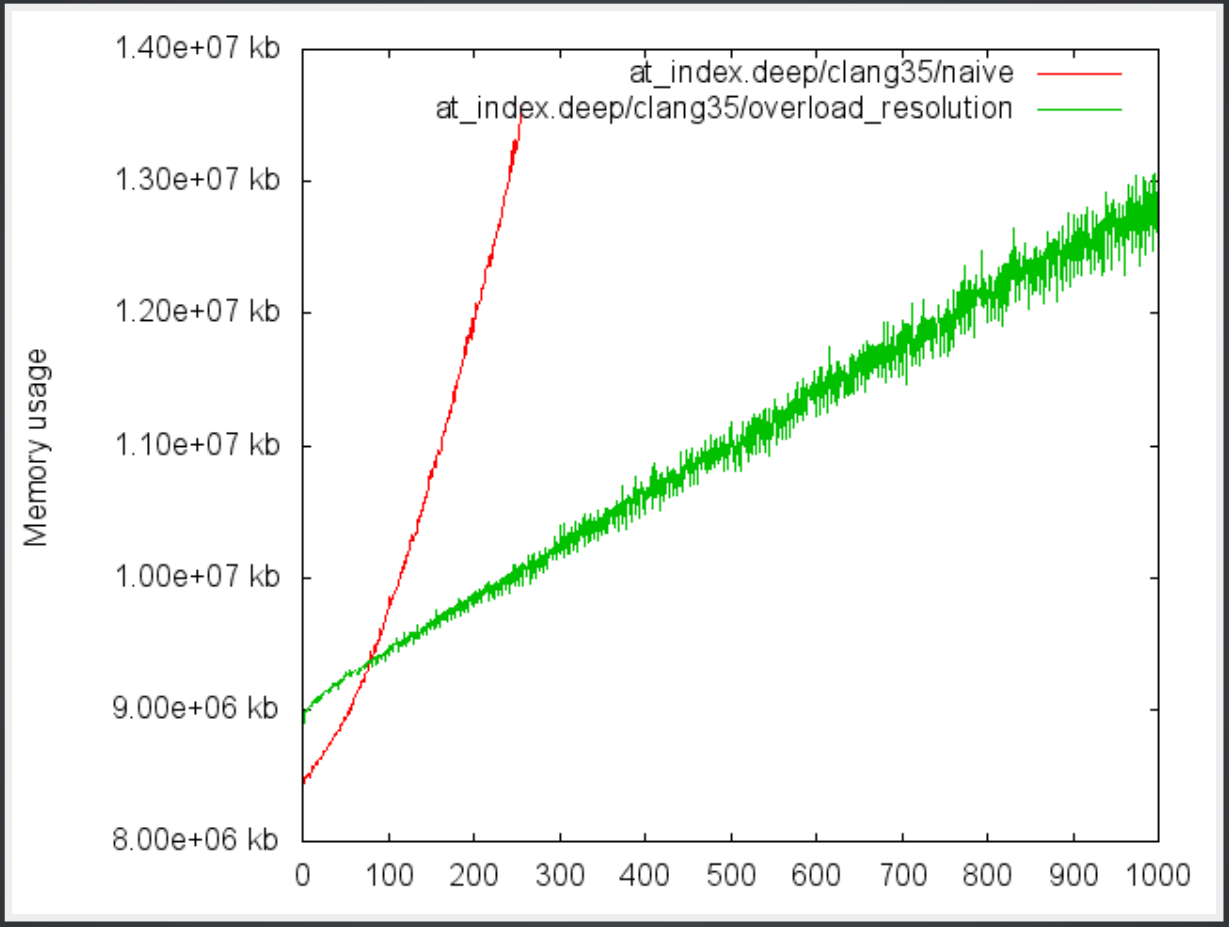
Using overload resolution

```
template <typename ignore>
struct lookup;

template <std::size_t ...ignore>
struct lookup<std::index_sequence<ignore...>> {
    template <typename nth>
    static no_decay<nth>
    apply(decltype(ignore, (void*)nullptr)..., no_decay<nth>*, ...);
};

template <std::size_t index, typename ...xs>
using at = decltype(
    lookup<std::make_index_sequence<index>>::apply(
        (no_decay<xs>*)nullptr...
    )
);
```

TIME


at_index.deep/clang35/naive
at_index.deep/clang35/overload_resolution
Compilation time
0.16 s
0.14 s
0.12 s
0.1 s
0.08 s
0.06 s
0.04 s
0.02 s
0 s
0
100
200
300
400
500
600
700
800
900
1000

MEMORY USAGE



LEFT-FOLDING

Naive

```
template <template <typename ...> class f, typename state, typename xs,
          bool = is_empty<xs>::value>
struct foldl {
    using type = state;
};

template <template <typename ...> class f, typename state, typename xs>
struct foldl<f, state, xs, false>
    : foldl<
        f,
        typename f<state, typename head<xs>::type>::type,
        typename tail<xs>::type
    >
{ };
```

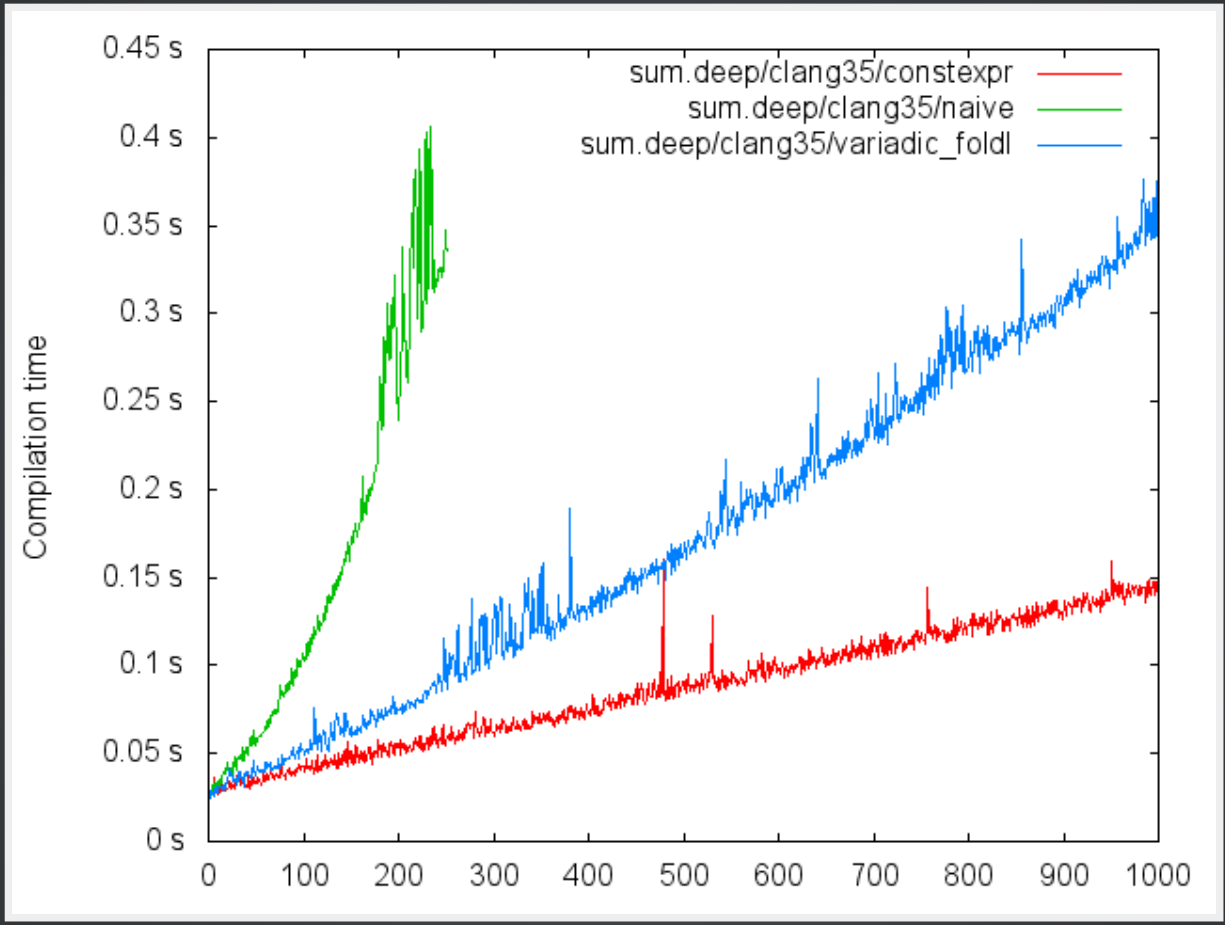
Using variadic templates

```
template <bool done> struct foldl_impl {
    template <template <class ...> class f, class state,
              class x, class ...xs>
    using result = typename foldl_impl<sizeof...(xs) == 0>::
        template result<f, typename f<state, x>::type, xs...>;
};

template <> struct foldl_impl<true> {
    template <template <class ...> class f, class state, class ...>
    using result = state;
};

template <template <class ...> class f, class state, class ...xs>
struct foldl {
    using type = typename foldl_impl<sizeof...(xs) == 0>::
                 template result<f, state, xs...>;
};
```

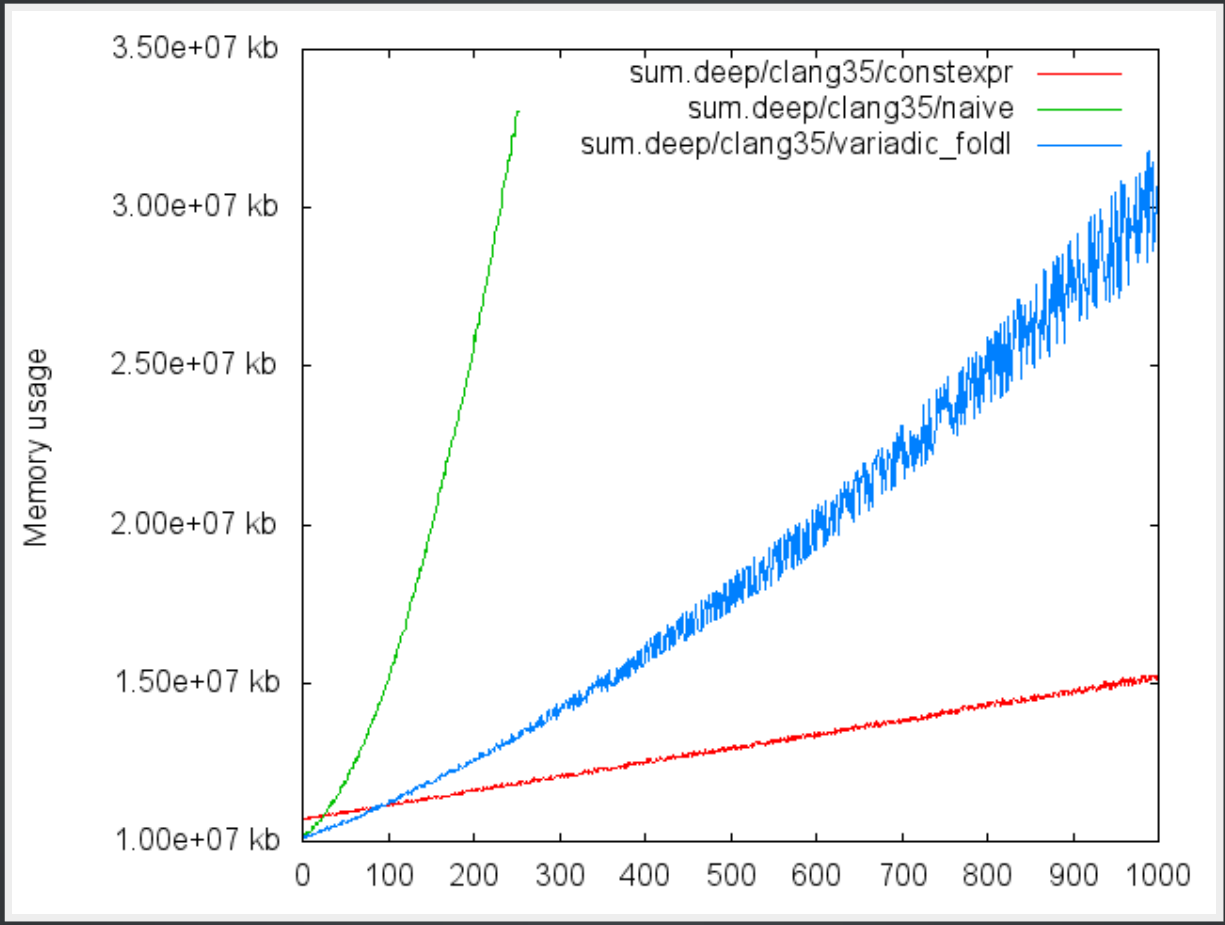
Bonus for specific folds on homogeneous data: `constexpr`

```
template <typename F, typename State, typename T, std::size_t N>
constexpr State
homogeneous_foldl(F f, State s, std::array<T, N> const& xs) {
    for (std::size_t i = 0; i < xs.size(); ++i)
        s = f(s, xs[i]);
    return s;
}

template <typename ...xs>
using sum = std::integral_constant<
    decltype(homogeneous_foldl(std::plus<>{}, 0, {xs::value...})),
    homogeneous_foldl(std::plus<>{}, 0, {xs::value...})
>;
```

TIME


sum.deep/clang35/constexpr
sum.deep/clang35/naive
sum.deep/clang35/variadic_foldl
Compilation time
0.45 s
0.4 s
0.35 s
0.3 s
0.25 s
0.2 s
0.15 s
0.1 s
0.05 s
0 s
0
100
200
300
400
500
600
700
800
900
1000

MEMORY USAGE



UNIVERSAL TEMPLATE TEMPLATE PARAMS

(or farewell `quoteN`)

C++03

```
template <template <typename> class f>
struct quote1 { ... };

template <template <typename, typename> class f>
struct quote2 { ... };

...

template <template <typename, typename, ..., typename> class f>
struct quoteN { ... };
```

C++11

```
template <template <typename ...> class f>
struct quote { ... };
```

AND THE LIST CONTINUES

http://github.com/ldionne/mpl11/tree/master/benchmarks

THE MPL IS STUCK WITH C++03
AND WE'RE STUCK WITH THE MPL

WE NEED SOMETHING NEW.
WHAT WOULD THAT LOOK LIKE?

ACT 2
TOWARDS A MPL SUCCESSOR

BUT BEFORE...
WHAT IS METAPROGRAMMING IN C++?

WHAT IS THE PURPOSE OF THE MPL?

OF FUSION?

ORTHOGONAL DESIGN ASPECTS

TAGGING SYSTEM

Basic tags (current MPL)

```
template <typename T>
struct tag_of { using type = typename T::mpl_tag; };

struct list_tag;

template <typename ...xs>
struct list { using mpl_tag = list_tag; };
```

Datatypes

```
template <typename T>
struct datatype { using type = typename T::mpl_datatype; };

struct List;

template <typename ...xs>
struct list { using mpl_datatype = List; };

template <typename x, typename xs>
struct cons { using mpl_datatype = List; };

struct nil { using mpl_datatype = List; };
```

METAFUNCTION DISPATCHING

Per-metafunction dispatching

```
template <typename Datatype> struct head_impl;

template <> struct head_impl<List> {
    template <typename> struct apply;

    template <typename x, typename ...xs>
    struct apply<list<x, xs...>> { using type = x; };

    template <typename x, typename xs>
    struct apply<cons<x, xs>> { using type = x; };
};

template <typename xs>
using head = typename head_impl<typename datatype<xs>::type>::
             template apply<xs>;

// etc...
```

Type classes (from Haskell)

```
template <> struct Iterable<List> : defaults<Iterable> {
    template <typename> struct head_impl;

    template <typename x, typename ...xs>
    struct head_impl<list<x, xs...>> { using type = x; };

    template <typename x, typename xs>
    struct head<cons<x, xs>> { using type = x; };

    // Same for tail<> and is_empty<>
};

template <typename xs>
using head = typename Iterable<typename datatype<xs>::type>::
             template head_impl<xs>;

// etc...
```

Type classes (cont.)

```
template <template <typename ...> class Typeclass>
struct defaults;

template <>
struct defaults<Iterable> {
    template <typename index, typename xs>
    struct at_impl { ... };

    template <typename index, typename xs>
    struct last_impl { ... };

    ...
};
```

SOME USEFUL TYPE CLASSES

(MPL11 type classes differ from Haskell's)

FUNCTOR

- `fmap<f, functor>`(m.c.d.)

FOLDABLE

- `foldl<f, state, foldable>`(m.c.d.)
- `foldr<f, state, foldable>`(m.c.d.)
- `sum<foldable>`
- `product<foldable>`
- `{any,all,none}<predicate, foldable>`
- `{maximum,minimum}<foldable>`
- …

ITERABLE

- `head<iterable>` (m.c.d.)
- `tail<iterable>` (m.c.d.)
- `is_empty<iterable>` (m.c.d.)
- `at<index, iterable>`
- `last<iterable>`
- `drop<n, iterable>`
- `drop_while<predicate, iterable>`
- …

ALGORITHM/SEQUENCE INTERACTION

Iterators

```
template <typename first, typename last>
struct algorithm_impl {
    // Use `next<>`, `deref<>`, `equal<>` and friends
};

template <typename sequence>
using algorithm = algorithm_impl<
    typename begin<sequence>::type,
    typename end<sequence>::type
>;
```

Sequence-oriented type classes

```
template <typename sequence>
struct algorithm {
    // Use appropriate type class methods.
};
```

EVALUATION STRATEGY

Strict with classic metafunctions

```
template <typename n, typename xs>
struct drop
    : if_<
        or_<
            typename equal<n, int_<0>>::type,
            typename is_empty<xs>::type
        >,
        xs,
        typename drop<
            typename minus<n, int_<1>>::type,
            typename tail<xs>::type
        >::type
    >
{ };
```

Strict with classic metafunctions (cont.)

```
template <typename n, typename xs,
    bool = or_<equal<n, int_<0>>, is_empty<xs>>::value>
struct drop {
    using type = xs;
};

template <typename n, typename xs>
struct drop<n, xs, false>
    : drop<
        typename minus<n, int_<1>>::type,
        typename tail<xs>::type
    >
{ };
```

Lazy with classic metafunctions

```
template <typename n, typename xs>
struct drop
    : if_<or_<equal<n, int_<0>>, is_empty<xs>>,
        xs,
        drop<minus<n, int_<1>>, tail<xs>>
    >
{ };
```

Strict with new-style metafunctions

```
template <typename N, typename Xs>
constexpr auto drop(N n, Xs xs) {
    return if_(n == int_<0>{},
        xs,
        drop(n - int_<1>{}, tail(xs))
    );
}
```

Strict with new-style metafunctions (cont.)

```
template <typename N, typename Xs>
constexpr auto drop_impl(N n, Xs xs, true_) { return xs; }

template <typename N, typename Xs>
constexpr auto drop(N n, Xs xs) {
    return drop_impl(n, xs, n == int_<0>{} || is_empty(xs));
}

template <typename N, typename Xs>
constexpr auto drop_impl(N n, Xs xs, false_) {
    return drop(n - int_<1>{}, tail(xs));
}
```

CONSIDERED DESIGNS

FAITHFUL MPL REIMPLEMENTATION

- Tagging: Basic
- Dispatching: Per-metafunction
- Algorithms: Iterators
- Evaluation: Strict + classic

HASKELL-ISH (MPL11)

- Tagging: Datatypes
- Dispatching: Type classes
- Algorithms: Type classes
- Evaluation: Lazy + classic

HETEROGENEOUS CONSTEXPR

- Tagging: Datatypes
- Dispatching: Type classes
- Algorithms: Type classes
- Evaluation: Strict + new-style

COMPARISON

METAFUNCTION MAPPING

MPL

```
using xs = vector<...>;
using ys = transform<xs, f>::type;
```

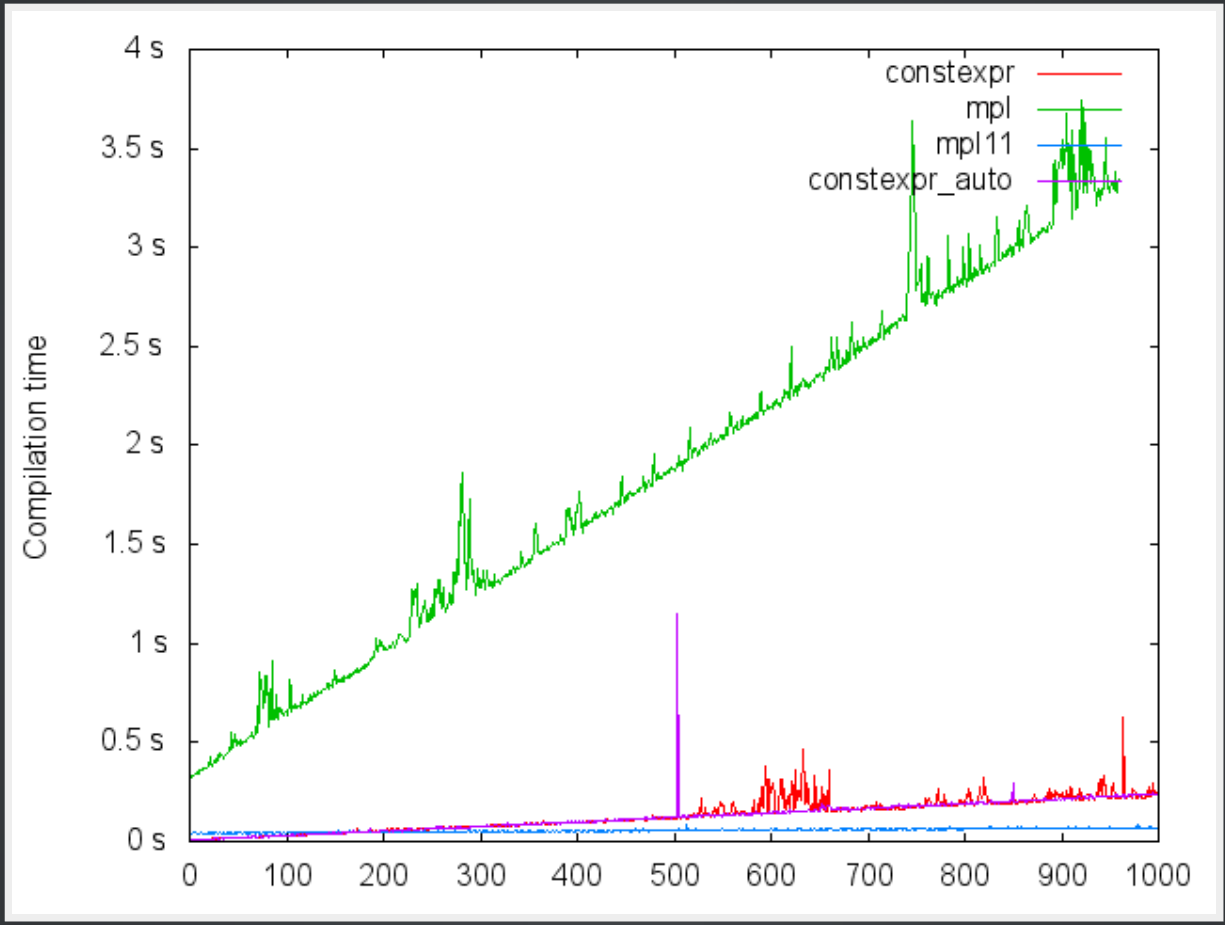
MPL11

```
using xs = list<...>;
using ys = fmap<f, xs>;
```

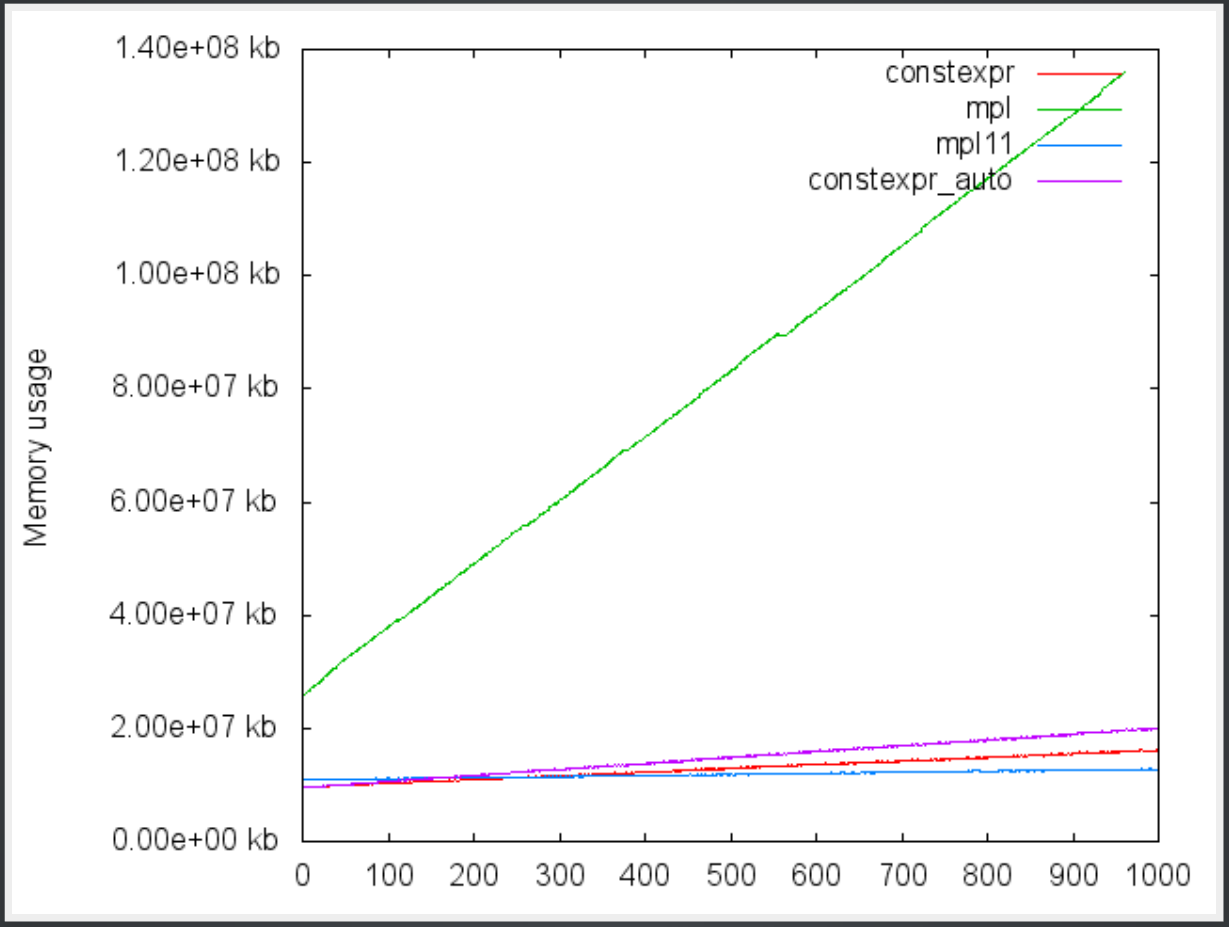
CONSTEXPR

```
constexpr auto xs = list<...>{};
constexpr auto ys = fmap(f, xs);
```

TIME


Compilation time
4 s
3.5 s
3 s
2.5 s
2 s
1.5 s
1 s
0.5 s
0 s
0
100
200
300
400
500
600
700
800
900
1000
constexpr
mpl
mpl11
constexpr_auto

MEMORY USAGE



LEFT FOLD

MPL

```
using xs = vector<...>;
using z = fold<xs, state, f>::type;
```

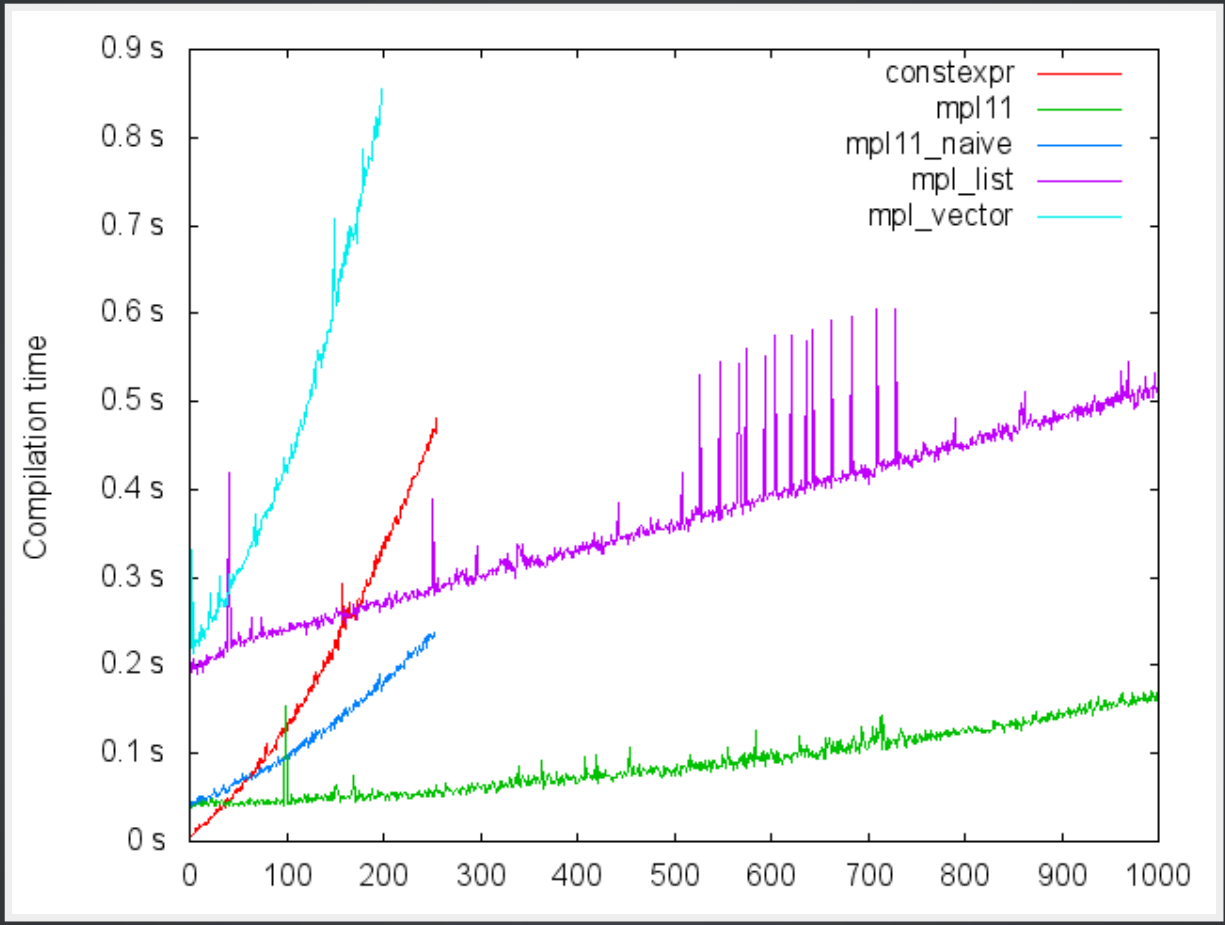
MPL11

```
using xs = list<...>;
using z = foldl<f, state, xs>;
```

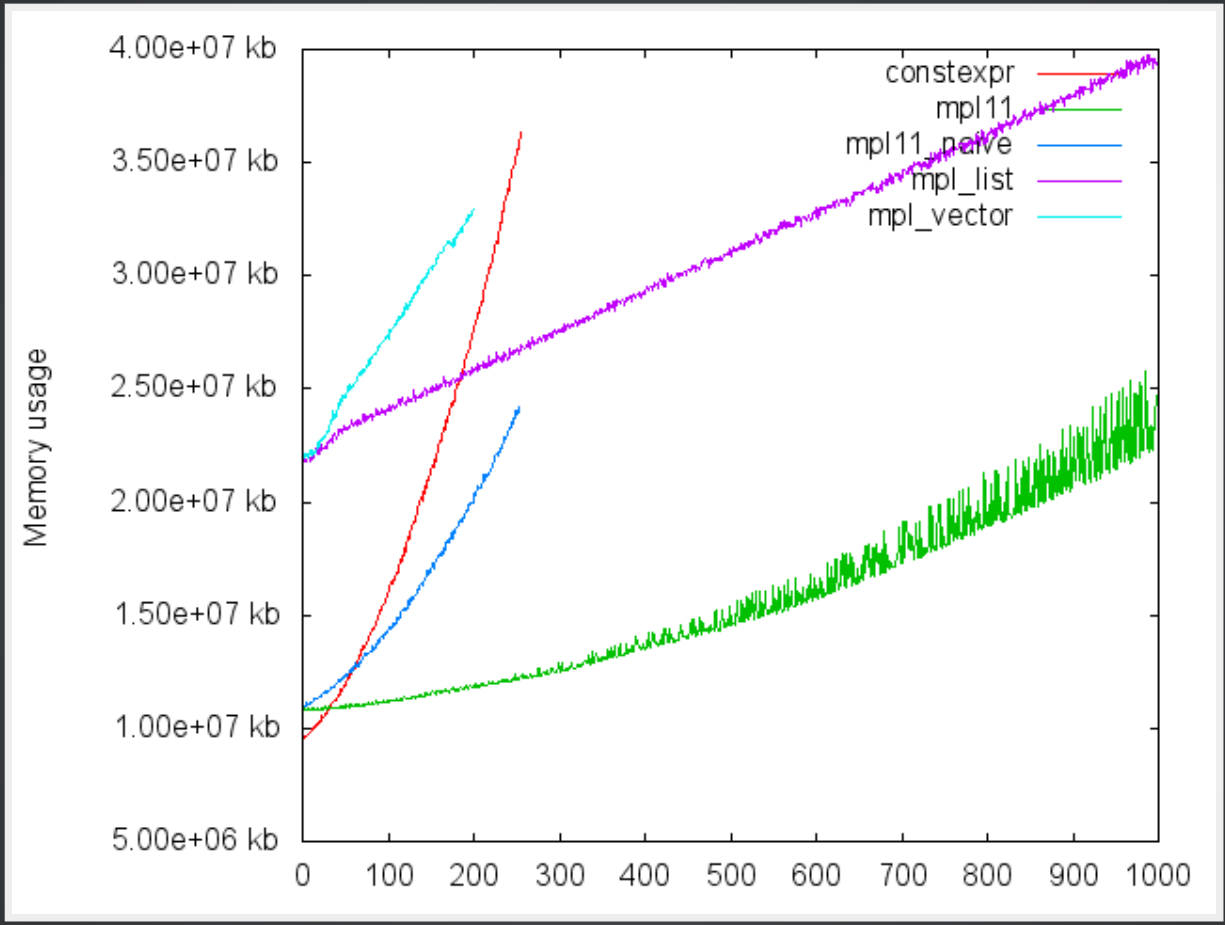
CONSTEXPR

```
constexpr auto xs = list<...>{};
constexpr auto ys = foldl(f, state, xs);
```

TIME


Compilation time
0.9 s
0.8 s
0.7 s
0.6 s
0.5 s
0.4 s
0.3 s
0.2 s
0.1 s
0 s
0
100
200
300
400
500
600
700
800
900
1000
constexpr
mpl11
mpl11_naive
mpl_list
mpl_vector

MEMORY USAGE



INCLUDING

GCC 4.9

- MPL: 8.2 s
- MPL11: 0.09 s
- MPL11 (minified): 0.08 s

GCC 4.9 (% SPEEDUP OVER THE MPL)

- MPL11: 9 111 %
- MPL11 (minified): 10 249 %

CLANG 3.5

- MPL: 2.6816 s
- MPL11: 0.0471 s

CLANG 3.5 (% SPEEDUP OVER THE MPL)

- MPL11: 5 693 %
- MPL11 (minified): 6 415 %

ROADMAP

GSOC

BOOST

STANDARD C++ (?)

THANK YOU

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

BONUS

(more TMP hacks)

LOGICAL AND

With noexcept

```
void allow_expansion(...) noexcept;

template <bool condition>
struct noexcept_if { noexcept_if() noexcept(condition) { } };

template <typename ...xs>
using and_ = bool_<
    noexcept(allow_expansion(noexcept_if<xs::value>{}...))
>;
```

With constexpr

```
template <std::size_t N>
constexpr bool and_impl(const bool (&array)[N]) {
    for (bool elem: array)
        if (!elem)
            return false;
    return true;
}

template <typename ...xs>
using and_ = bool_<
    and_impl<sizeof...(xs) + 1>({(bool)xs::value..., true})
>;
```

With overload resolution

```
template <typename ...T> true_  pointers_only(T*...);
template <typename ...T> false_ pointers_only(T...);
                         true_  pointers_only();

template <typename ...xs>
using and_ = decltype(pointers_only(
    std::conditional_t<xs::value, int*, int>{}...
));
```

With partial specialization

```
template <typename ...>
struct and_impl : false_ { };

template <typename ...T>
struct and_impl<std::integral_constant<T, true>...> : true_ { };

template <typename ...xs>
using and_ = and_impl<bool_<xs::value>...>;
```

INDEX-BASED LOOKUP

Using multiple inheritance

```
template <std::size_t index, typename value>
no_decay<value> lookup(index_pair<index, value>*);

template <typename indices, typename ...xs>
struct index_map;

template <std::size_t ...indices, typename ...xs>
struct index_map<std::index_sequence<indices...>, xs...>
    : index_pair<indices, xs>...
{ };

template <std::size_t index, typename ...xs>
using at = decltype(lookup<index>(
    (index_map<std::index_sequence_for<xs...>, xs...>*)nullptr
));
```

Using multiple inheritance (v2)

```
template <std::size_t, std::size_t, typename x>
struct select { };

template <std::size_t n, typename x>
struct select<n, n, x> { using type = x; };

template <std::size_t n, typename indices, typename ...xs>
struct lookup;

template <std::size_t n, std::size_t ...index, typename ...xs>
struct lookup<n, std::index_sequence<index...>, xs...>
    : select<n, index, xs>...
{ };

template <std::size_t n, typename ...xs>
using at = lookup<n, std::index_sequence_for<xs...>, xs...>;
```