Instantiations Must Go

(continued)

High Order Functions

- Fold
- Bind
- Transform
- Lambda Expressions

Fold

```
extern struct fold_
{
  template< class Fn, class St, class Seq >
  auto operator()(type_<Fn>& fn, type_<St>& st, type_<Seq>& s) -> decltype
  (
    fold( fn
        , apply( fn, st, front( s ) )
        , pop_front( s )
        )
  );

  template< class Fn, class St, class Seq >
  auto operator()(Fn& fn, type_<St>& st, type_<Seq>& s) -> decltype
  (
    st
  );
} & fold;
```

Bind

```
extern struct bind_
{
  template< class Fn, class... Args >
  auto operator()(type_<Fn>& fn, type_<Args>&...args) -> decltype
  (
    type< bind_(type_<Fn>&, type_<Args>&...) >()
  );
} & bind;

extern struct apply_
{
  template<class Fn, class... Args>
  auto operator()(type_<Fn>& fn, type_<Args>&... args) -> decltype
  (
    apply_impl(type<Fn>(), type<Args>()...)
  );
} & apply;

template<class Fn> Fn& make(type_<Fn>&);

template<class Fn, class... Args>
auto apply_impl(type_<Fn>& fn, type_<Args>&... args) -> decltype
(
  make(type<Fn>())(type<Args>()...)
);
```

Bind (continued)

```
template<class Fn, class... BindArgs, class... Args>
auto apply_impl(type_<bind_(type_<Fn>&, type_<BindArgs>&...)>&, type_<Args>&...)
-> decltype(
  bind_impl(type<bind_(type_<Fn>&, type_<BindArgs>&...)>(), type<Args>()...)
);

template<class t, class...args>
auto bind_impl(type_<t>& obj, type_<args>&...) -> decltype
(
  obj
);

template<class Fn, class... BindArgs, class... Args>
auto bind_impl(type_<bind_(type_<Fn>&, type_<BindArgs>&...)>&, type_<Args>&... args)
-> decltype(
  apply( bind_impl(type<Fn>(), args...)
       , bind_impl(type<BindArgs>(), args...)...
       )
);

template<int N> struct arg_;

template<int N> auto arg() -> decltype(type<arg_<N> >());
```

Bind (continued)

```
template<class Front, class...Rest>
auto bind_impl(type_<arg_<1> >&, type_<Front>& fr, type_<Rest>&...) -> decltype
(
  fr
);

template<int N, class Front, class...Rest>
auto bind_impl(type_<arg_<N> >&, type_<Front>&, type_<Rest>&... args) -> decltype
(
  bind_impl(arg<N-1>(), args...)
);

extern type_<arg_<1> > & _1;
extern type_<arg_<2> > & _2;
extern type_<arg_<3> > & _3;
extern type_<arg_<4> > & _4;
extern type_<arg_<5> > & _5;
```

Transform

```
extern struct transform_
{
  template <class Fn, class Seq>
  auto operator()(type_<Fn>& fn, type_<Seq>& s) -> decltype
  (
    fold( bind( type( back_inserter ), _1, bind( fn, _2 ) ) )
        , vector()
        , s
        )
  );
} & transform;
```

MPL Vs. Our Approach

- MPL Syntax

```
front
< push_back< vector< int, float >
           , long long
           >::type
>::type
```

MPL Vs. Our Approach

- MPL Syntax (<> unfamiliar for functions, ::type)

```
front
< push_back< vector< int, float >
           , long long
           >::type
>::type
```

MPL Vs. Our Approach

- MPL Syntax (<> unfamiliar for functions, ::type)

```
front
< push_back< vector< int, float >
           , long long
           >::type
>::type
```

- Our Syntax

```
front
( push_back( vector( int_(), float_() )
           , long_long()
           )
)
```

MPL Vs. Our Approach

- MPL Syntax (<> unfamiliar for functions, ::type)

```
front
< push_back< vector< int, float >
           , long long
           >::type
>::type
```

- Our Syntax (consistent with functions, needs wrappers)

```
front
( push_back( vector( int_(), float_() )
           , long_long()
           )
)
```

MPL Vs. Our Approach

- MPL Syntax

```
typename front
< typename push_back< vector< Arg, float >
                    , long long
                    >::type
>::type
```

MPL Vs. Our Approach

- MPL Syntax (everybody hates typename)

```
typename front
< typename push_back< vector< Arg, float >
                    , long long
                    >::type
>::type
```

MPL Vs. Our Approach

- MPL Syntax (everybody hates typename)

```
typename front
< typename push_back< vector< Arg, float >
                    , long long
                    >::type
>::type
```

- Our Syntax

```
front
( push_back( vector( arg, float_() )
           , long_long()
           )
)
```

MPL Vs. Our Approach

- MPL Syntax (everybody hates typename)

```
typename front
< typename push_back< vector< Arg, float >
                    , long long
                    >::type
>::type
```

- Our Syntax (*yawn*)

```
front
( push_back( vector( arg, float_() )
           , long_long()
           )
)
```

Current Libraries

STL:

- Run-time homogenous containers and iterators
- Run-time iterators
- Run-time algorithms

Current Libraries

STL:

- Run-time homogenous containers and iterators
- Run-time iterators
- Run-time algorithms

Boost.MPL:

- Compile-time versions of containers
- Compile-time versions of algorithms
- Compile-time lambda expressions
- Modeled after the STL
- Limited run-time interaction (boost::mpl::for_each)

Current Libraries

STL:

- Run-time homogenous containers and iterators
- Run-time iterators
- Run-time algorithms

Boost.MPL:

- Compile-time versions of containers
- Compile-time versions of algorithms
- Compile-time lambda expressions
- Modeled after the STL
- Limited run-time interaction (boost::mpl::for_each)

Boost.Fusion:

- Partially run-time heterogenous containers
- Models some Boost.MPL concepts
- Algorithms for heterogenous containers
- Operates on types *and* run-time values

Current Libraries

STL:

- Run-time homogenous containers and iterators
- Run-time iterators
- Run-time algorithms

~~Boost.MPL:~~

- ~~Compile-time versions of containers~~
- ~~Compile-time versions of algorithms~~
- ~~Compile-time lambda expressions~~
- ~~Modeled after the STL~~
- ~~Limited run-time interaction (boost::mpl::for_each)~~

Is This Redundant?

Boost.Fusion:

- Partially run-time heterogenous containers
- Models some Boost.MPL concepts
- Algorithms for heterogenous containers
- Operates on types *and* run-time values

MPL Vs. Our Approach (transform)

- MPL Syntax

```
transform
< vector_c< int, 1, 2, 3 >
, vector_c< int, 4, 5, 6 >
, plus< _1, _2 >
>::type
```

MPL Vs. Our Approach (transform)

- MPL Syntax (named “plus”)

```
transform
< vector_c< int, 1, 2, 3 >
, vector_c< int, 4, 5, 6 >
, plus< _1, _2 >
>::type
```

MPL Vs. Our Approach (transform)

- MPL Syntax (named "plus")

```
transform
< vector_c< int, 1, 2, 3 >
, vector_c< int, 4, 5, 6 >
, plus< _1, _2 >
>::type
```

- Our Syntax

```
transform
( vector_c< int, 1, 2, 3 >()
, vector_c< int, 4, 5, 6 >()
, _1 + _2
)
```

MPL Vs. Our Approach (transform)

- MPL Syntax (named “plus”)

```
transform
< vector_c< int, 1, 2, 3 >
, vector_c< int, 4, 5, 6 >
, plus< _1, _2 >
>::type
```

- Our Syntax (Boost.Fusion and Boost.Phoenix!)

```
transform
( vector_c< int, 1, 2, 3 >()
, vector_c< int, 4, 5, 6 >()
, _1 + _2
)
```

vector_c is a loose wrapper around Boost.Fusion!

```
template< class ValueType, ValueType... Value >
auto vector_c() -> decltype
(
  fusion::make_vector( integral_c< ValueType, Value >()... )
);
```

vector_c is a loose wrapper around Boost.Fusion!

```
template< class ValueType, ValueType... Value >
auto vector_c() -> decltype
(
  fusion::make_vector( integral_c< ValueType, Value >()... )
);
```

Transform is Boost.Fusion transform!

Lambda expressions are a modified Boost.Phoenix!

A Note about Boost.Fusion

Why is MPL implemented as it is?

Taking Fusion further...

```
BOOST_FTMPL_EVAL
( at_c< 2 >
  ( transform
    ( vector( type< int >(), type< float >(), type< double >() )
    , add_pointer
    )
  )
)
```

Overall Goals

- A single, unified library for MPL and Fusion
- Generic at the lowest possible level
- No need to overload specifically for metaprogramming
- Zero runtime penalty
- No redundancy
- Minimal compile-time penalty

Overall Goals

- A single, unified library for MPL and Fusion
- Generic at the lowest possible level
- No need to overload specifically for metaprogramming
- Zero runtime penalty
- No redundancy
- Minimal compile-time penalty

Well, one can dream...

Unifying MPL and Fusion: What We Need...

Generic Algorithms:

- fold
- transform
- for_each
- find
- count
- etc.

Unifying MPL and Fusion: What We Need...

Generic Algorithms:

- fold
- transform
- for_each
- find
- count
- etc.

Concepts:

???

Unifying MPL and Fusion: What We Need...

Generic Algorithms:

- fold
- transform
- for_each
- find
- count
- etc.

Concepts:

- Boost.Fusion's already got this covered!

Updating our approach, Fusion and Phoenix

- Rewrite Fusion to not depend on MPL
- Make metafunctions valid function objects
- Update Phoenix to internally use `decltype`
- Wrap types when metaprogramming
- Make traits of models of Fusion concepts yield Fusion objects

```
template< class... Type >
class vector
{
  typedef vector< decltype( type< Type >() )... > types;
  static types& types_();
  // remaining vector implementation...
};
```

A sequence of the types in a Fusion sequence is another Fusion sequence...

But is it worth it? What are the downsides?

- Requires a C++0x compiler
- Partial rewrite of Boost.Fusion
- MPL as it is is widely used
- Boost.Type_Traits consists of MPL metafunctions
- MPL is more lightweight than Boost.Fusion
- We didn't decrease compile-times (*clang, save us*)

Questions