Boost.Dispatch
A Generic Tag-Dispatching Library

metascale
Software tools for high-performance computing

Joel Falcou – Mathias Gaunard

23 mai 2013

Motivation and Scope

Generic Programming

- Optimizations through Specialisations
- ... but how to specialize ?
- What we want is Concepts (based overloads)

Motivation and Scope

Generic Programming

- Optimizations through Specialisations
- ... but how to specialize ?
- What we want is Concepts (based overloads)

Introducing Boost.Dispacth

- Generic way to handle specializations and related optimizations
- Minimize code duplication by an expressive definition of types constraints
- Increase applicability of Tag Dispatching

What's in this talk ?

Overloads, SFINAE, Tag Dispatching, Oh My ...

- Why overloads in C++ are that useful
- Getting further with SFINAE
- Tag Dispatching Unplugged

What's in this talk ?

Overloads, SFINAE, Tag Dispatching, Oh My ...

- Why overloads in C++ are that useful
- Getting further with SFINAE
- Tag Dispatching Unplugged

Introducing Boost.Dispatch

- Motivation and Rationale
- The Generic Hierarchy System
- The Generic Function Caller
- Unusal Hierarchies
- Trivial and non-trivial use cases

Disclaimer

This talk may contain traces of Boost.Proto

Disclaimer

This talk may contain traces of Boost.Proto

Not that much really

Function Overloading Rules

General Process [1]

- The name is looked up to form an initial *Overload Set*.
  If necessary, this set is tweaked in various ways.
- Any candidate that doesn't match the call at all is eliminated from the overload set, building the *Viable Set*.
- Overload resolution is performed to find the *Best Viable Function*.
- The selected candidate is checked and potential diagnostic is issued.

[1] *C++ Templates: The Complete Guide* – David Vandevoorde, Nicolai M. Josuttis

Function Overloading Rules

General Process [1]

- The name is looked up to form an initial *Overload Set*.
  If necessary, this set is tweaked in various ways.
- Any candidate that doesn't match the call at all is eliminated from the overload set, building the *Viable Set*.
- Overload resolution is performed to find the *Best Viable Function*.
- The selected candidate is checked and potential diagnostic is issued.

What to do with that ?

- What are the rule for building the Overload Set (Ω)?
- How to define the "Best Candidate" ?

[1] *C++ Templates: The Complete Guide* – David Vandevoorde, Nicolai M. Josuttis

All Glory to the Overload Set

Building Ω

- Add all non-template functions with the proper name
- Add all template functions once template resolution is successful

Notes

- Ω is a lattice: non-template supersede template functions
- We need to refine what a success means for template functions
- All of this use ADL if needed

Finding `nemo()`

Best Viable Function selection process

- Determine the Implicit Conversion Sequence (ICS) for each arguments
- Categorize and rank them
- If any argument fails this process, compiler frowns.

Finding `nemo()`

The Implicit Conversion Sequence (ICS)

- Standard conversion sequences
  - Exact match
  - Promotion
  - Conversion
- User-defined conversion sequences defined as:
  - A standard conversion sequence
  - A user-defined conversion
  - A second standard conversion sequence
  - An UDCS is better than an other if it has the same UDC but a better second SCS
- Ellipsis conversion sequences

assert(Mind==Blown)

Small example

```
void f(int)         { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double)      { cout << "void f(double)\n"; }

int main()
{
  f(1); f(1.); f("1");  f(1.f);  f('1');
}
```

Output

- `f(1)` → `void f(int)`
- `f(1.)` → `void f(double)`
- `f("1")` → `void f(char const*)`
- `f(1.f)` → `void f(double)`
- `f('1')` → `void f(int)`

assert(Mind==Blown)

Small example

```
void f(int)          { cout << "void f(int)\n"; }
void f(char const*) { cout << "void f(char const*)\n"; }
void f(double)       { cout << "void f(double)\n"; }
template<class T> void f(T) { cout << "void f(double)\n"; }

int main()
{
  f(1); f(1.); f("1");  f(1.f);  f('1');
}
```

Output

- `f(1)` → `void f(int)`
- `f(1.)` → `void f(double)`
- `f("1")` → `void f(char const*)`
- `f(1.f)` → `void f(T)`
- `f('1')` → `void f(T)`

Substitution Failures Are What ???

```
template<typename Container>
typename Container::size_type f(Container const&)
{
  return c.size();
}

int main()
{
  std::vector<double> v(4);
  f(v);

  f(1); /// OMG Incoming Flaming Errors of Doom
}
```

Substitution Failures Are What ???

```
template<typename Container>
typename Container::size_type f(Container const&)
{
  return c.size();
}

int main()
{
  std::vector<double> v(4);
  f(v);

  f(1); /// OMG Incoming Flaming Errors of Doom
}
```

```
error:  no matching function for call to 'f(int)'
```

Substitution Failures Are What ???

Definition

- We want generate Ω for a given function
- Some of the candidates functions are result of a template substitution
- If this substitution fails, the function is removed from Ω and no error are emited
- If at Ω ends up non ambiguous and not empty, we proceeed to the next step

SFINAE in practice - Rebuilding enable_if

```
template<bool Condition, typename Result = void>
struct enable_if;

template<typename Result>
struct enable_if<true,Result>
{
  typedef Result type;
};
```

SFINAE in practice - Rebuilding enable_if

```
template<typename T>
typename enable_if<(sizeof(T)>2)>::type
f( T const& )
{
  cout << "That's a big type you have there !\n";
}

template<typename T>
typename enable_if<(sizeof(T)<=2)>::type
f( T const& )
{
  cout << "Oooh what a cute type!\n";
}
```

SFINAE in practice - The dreadful enable_if_type

```
template<typename Type, typename Result = void>
struct enable_if_type
{
  typedef Result type;
};

template<typename T, typename Enable = void> struct size_type
{
  typedef std::size_t type;
};


template<typename T> struct size_type<T,typename
    enable_if_type<typename T::size_type>::type
{
  typedef typename T::size_type type;
};
```

SFINAE in practice - Type traits definition

```
template<typename T>
struct is_class
{
  typedef char yes_t
  typedef struct { char a[2]; } no_t;

  template<typename C> static yes_t test(int C::*);
  template<typename C> static no_t test(...);

  static const bool value = sizeof(test<T>(0)) == 1;
};
```

Tag Dispatching

Limitation of SFINAE

- Conditions must be non-overlapping
- Difficult to extend
- Compilation is $O(N)$ with number of cases

Principles of Tag Dispatching

- Categorize family of type using a tag hierarchy
- Easy to extend : add new category and/or corresponding overload
- Uses overloading rules to select best match
- Poor man's Concept overloading

Tag Dispatching - std::advance

```
namespace std
{
  struct input_iterator_tag {};
  struct bidirectional_iterator_tag : input_iterator_tag {};
  struct random_access_iterator_tag
      : bidirectional_iterator_tag {};
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class InputIterator, class Distance>
    void advance_dispatch( InputIterator& i
                         , Distance n
                         , input_iterator_tag const&
                         )
    {
      assert(n >=0);
      while (n--) ++i;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class BidirectionalIterator, class Distance>
    void advance_dispatch( BidirectionalIterator& i
                         , Distance n
                         , bidirectional_iterator_tag const&
                         )
    {
      if (n >= 0)
        while (n--) ++i;
      else
        while (n++) --i;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  namespace detail
  {
    template <class RandomAccessIterator, class Distance>
    void advance_dispatch( RandomAccessIterator& i
                         , Distance n
                         , random_access_iterator_tag const&
                         )
    {
      i += n;
    }
  }
}
```

Tag Dispatching - std::advance

```
namespace std
{
  template <class InputIterator, class Distance>
  void advance(InputIterator& i, Distance n)
  {
    typename iterator_traits<InputIterator>::iterator_category
         category;
    detail::advance_dispatch(i, n, category);
  }
}
```

Boost.Dispatch

From NT2, Boost.SIMD to Boost.Dispatch

- NT2 and Boost.SIMD use fine grain function overload for performances reason
- Problem was : NT2 is 500+ functions over 10+ architectures
- How can we handle this amount of overloads in an **extensible way** ?

Our Goals

- Provide a generic entry point for tag dispatching
- Provide base hierarchy tags for useful types (including Fusion and Proto types)
- Provide a way to categorize functions and architecture properties
- Provide a generic "*dispatch me this*" process

Boost.Dispatch - Hierarchy

The Hierarchy Concept

`H` models `Hierarchy` if

- `H` inherits from another `Hierarchy P`
- `H::parent` evaluates to `P`
- Usually Hierarchy are template types carrying the type they hierarchize
- Hierarchy are topped by an `unspecified_<T>` hierarchy

Boost.Dispatch - Hierarchy

```
template<typename I>
struct input_iterator_tag : unspecified_<I>
{
  typedef unspecified_<I> parent;
};

template<typename I>
struct bidirectional_iterator_tag : input_iterator_tag _<I>
{
  typedef input_iterator_tag _<I> parent;
};

template<typename I>
struct random_access_iterator_tag
     : bidirectional_iterator_tag _<I>
{
  typedef bidirectional_iterator_tag _<I> parent;
};
```

Boost.Dispatch - `hierarchy_of`

How to access hierarchy of a given type ?

- `hierarchy_of` is a meta-function giving you the hierarchy of a type
- `hierarchy_of` is extendable by specialization or SFINAE
- Currently tied to NT2 view of things

Example - `f(T t)`

- returns `1/t` if it's a floating point value
- returns `-t` if it's a signed integral value
- returns `t` otherwise

Boost.Dispatch - hierarchy_of example

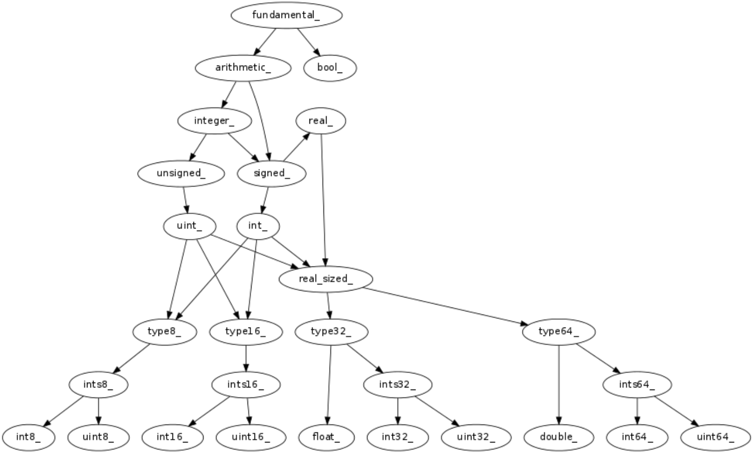
```
template<typename T> T f_( T const& t, scalar_<real_<T>> )
{
  return T(1)/t;
}

template<typename T> T f_( T const& t, scalar_<signed_<T>> )
{
  return -t;
}

template<typename T> T f_( T const& t
                         , scalar_<unspecified_<T>> )
{
  return t;
}

template<typename T> T f( T const& t )
{
  return f_(t, hierarchy_of<T>::type());
}
```

Boost.Dispatch - The basic types hierarchy



Boost.Dispatch - The basic types hierarchy

Register types informations

- Basic types hierarchy is built on top of one of the previous properties
- If the types is regular, its hierarchy is wrapped by `scalar_<.>`
- If not,special wrappers are used.
- Both `scalar_<.>` and other wrapper goes into `generic_<.>`

Application

- `simd_<.>` helps hierarchizing native SIMD types
- If you have code looking the same for scalar and SIMD, dispatch on `generic_<.>`
- One can think of having stuff like `vliw_<.>` or `proxy_<.>` wrappers

Boost.Dispatch - Other useful hierarchies

Array and Fusion Sequence

- fusion_sequence<T> hierarchizes all Fusion Sequence type
- array<T,N> hierarchizes all array types
- array<T,N> is obviously a sub-hierarchy from fusion_sequence<T>

Proto Expressions

- expr_<T,Tag,N>is a proto AST with all informations available
- node_<T,Tag,N,D> is a proto AST on which the tag is hierarchized
- ast_<T,D> represents any Proto AST.

Boost.Dispatch - Gathering functions properties

Our Motivation

- Parallel code can be refactored using Parallel Skeletons
- A lot of functions share implementations
- How can we know about a function properties ?

Boost.Dispatch - Gathering functions properties

Our Motivation

- Parallel code can be refactored using Parallel Skeletons
- A lot of functions share implementations
- How can we know about a function properties ?

Solution : Functions Tag

- Associate a type to each function
- Give a hierarchy to this tag
- Make those functions hierarchy useful

Boost.Dispatch - Function Tag examples

```
template<class Tag, class U, class B, class N>
struct reduction_ : unspecified_<Tag> { ... };

template<class Tag>
struct elementwise_ : unspecified_<Tag> { ... };

struct plus_ : elementwise_<plus_> { ... };

struct sum_ : reduction_<sum_, sum_, plus_, zero_> { ... };
```

Boost.Dispatch - Gathering architectures properties

Our Motivation (again)

- Drive optimization by knowledge of the architecture
- Embed this knowledge into the dispatching system
- Allow for architecture description to be derived

Solution : Architectural Tag

- All traditional architectural element is a tag
- Those tag can be compound Hierarchy
- Make function use the current architecture tag as an additional hidden parameters

Boost.Dispatch - Architecture Tag examples

```
struct formal_ : unspecified_<formal_> { ... };
struct cpu_ : formal_ { ... };

template<class Arch>
struct cuda_ : Arch { ... };

template<class Arch>
struct openmp_ : Arch { ... };

struct sse_    : simd_   {};
struct sse2_   : sse_    {};
struct sse3_   : sse2_   {};
struct sse4a_  : sse3_   {};
struct sse4_1_ : ssse3_  {};
struct sse4_2_ : sse4_1_ {};
struct avx_    : sse4_2_ {};

// Architecture usign openMP on an avx CPU
// and equipped with a CUDA enabledGPU
typedef gpu_< opemp_< avx_ > > my_arch;
```

Boost.Dispatch - Putting it together

dispatch_call

- Gather information about the function and the architecture
- Computes the hierarchization of function parameters
- Dispatch to an externally defined implementation

functor

- Generic tag based functor
- Encapsulate `dispatch_call` calls
- TR1 compliant functor

Boost.Dispatch - plus

```
template<typename A, typename B>
auto plus(A const& a, B const& b)
  ->  decltype(dispatch_call< plus_(A const&, B const&)
                            , typename default_site<plus_>::
                              type
                            >::type()(a,b)
              )
{
  return typename dispatch_call< plus_(A const&, B const&)
                               , typename default_site<plus_
                                 >::type
                               >::type()(a,b);
};
```

Boost.Dispatch - plus

```
template<typename A, typename B>
auto plus(A const& a, B const& b)
  -> decltype(functor<plus_>()(a,b))
{
  functor<plus_> callee;
  return calle(a,b);
};
```

Boost.Dispatch - plus

```
BOOST_SIMD_FUNCTOR_IMPLEMENTATION( plus_,cpu_, (A0)
                              , ((scalar_<unspecified_<A0>>))
                                ((scalar_<unspecified_<A0>>))
                                  )
{
  auto operator()(A0 const& a, A0 const& b) const
       -> decltype(a+b)
  {
    return a+b;
  }
};
```

Boost.Dispatch - plus

```
BOOST_SIMD_FUNCTOR_IMPLEMENTATION( plus_,sse2_, (A0)
                                 , ((simd_<double_<A0>,sse_>))
                                   ((simd_<double_<A0>,sse_>))
                                 )
{
  __m128d operator()(__m128d a, __m128d b) const
  {
    return _mm_add_pd( a, b );
  }
};
```

Boost.Dispatch - plus

```
BOOST_SIMD_FUNCTOR_IMPLEMENTATION( plus_ ,cpu_ , (A0)(A1)
                                 , (fusion_sequence<A0>)
                                   (fusion_sequence<A1>)
                                 )
{
  auto operator()(A0 const& a, A1 const& b) const
    -> decltype(fusion::transform(a,b, functor<plus_>()))
  {
    return fusion::transform(a,b, functor<plus_>());
  }
};
```

Boost.Dispatch - plus

```
BOOST_SIMD_FUNCTOR_IMPLEMENTATION
      ( plus_, formal_, (D)(A0)(A1)
      , ((node_<A0,multiplies_,long_<2>,D>))
        (unspecified_<A1>)
      )
{
  BOOST_DISPATCH_RETURNS(2, (A0 const& a0, A1 const& a1),
    fma( boost::proto::child_c<0>(a0)
       , boost::proto::child_c<1>(a0)
       , a1
       )
  )
};
```

Boost.Dispatch - NT2 E.T operations

```
NT2_FUNCTOR_IMPLEMENTATION( transform_, cpu_
                          , (A0)(A1)(A2)(A3)
                          , ((ast_<A0,domain>))
                            ((ast_<A1,domain>))
                            (scalar_<integer_<A2>>)
                            (scalar_<integer_<A3>>)
                          )
{
  void operator()(A0& a0, A1& a1, A2 p, A3 sz) const
  {
    typedef typename A0::value_type stype;
    for(std::size_t i=p; i != p+sz; ++i)
      nt2::run(a0, i, nt2::run(a1, i, meta::as_<stype>()));
  }
};
```

Boost.Dispatch - NT2 E.T operations

```
NT2_FUNCTOR_IMPLEMENTATION( transform_, openmp_<Site>
                          , (A0)(A1)(Site)(A2)(A3)
                          , ((ast_<A0, domain>))
                            ((ast_<A1, domain>))
                            (scalar_< integer_<A2> >)
                            (scalar_< integer_<A3> >)
                          )
{
  void operator()(A0& a0, A1& a1, A2 it, A3 sz) const
  {
    nt2::functor<tag::transform_,Site> transformer;
    auto size = sz/threads(), over = sz%threads();
    #pragma omp parallel for
    for(std::ptrdiff_t p=0;p<threads();++p)
    {
      auto offset = size*p + std::min(over,p);
      size += over ? ((over > p) ? 1 : 0) : 0;
      transformer(a0,a1,it+offset,size);
    }
  }
};
```

Wrapping this up

Tag Dispatching as a Tool

- Good surrogate for Concept overloading
- Scalable compile-time wise
- Applicable with success to a lot of situations

Boost.Dispatch

- Tag Dispatching on steroids
- Function/Architecture Tag open up design space
- Easy to extend and modularize

Future Works

Availability

- Currently lives as a subcomponent of Boost.SIMD
- Play with it from `https://github.com/MetaScale/nt2`
- Opinions/Tests welcome

Remaining Challenges

- Compile-time improvement
- More generalization for `hierarchy_of`
- Make it works on more compilers
- Submission to Boost review

Thanks for your attention !