An experimental DSL for template metaprogramming

Dan Marsden

May 1, 2009

Outline

Introduction

This talk is about

- An experimental tool for working with Boost.MPL
- An unusual perspective on the MPL
- DSL development outside our familiar C++ world

Ground rules

Todays talk

- Minimal Haskell - it's not a Haskell talk
- MPL concepts will be introduced - although quickly
- I'll aim for plenty of examples

Outline

Assembler level template metaprogramming

```
template<bool b, typename T,
         template<typename T> class F>
struct my_type {
  typedef T result;
};

template<typename T,
         template<typename T> class F>
template
struct my_type<true> {
  typedef typename F<T>::type result;
};
```

The wonder of libraries

```
template<typename B, typename T, typename F>
struct my_metafunction {
  typedef typename mpl::eval_if<
   B,
   mpl::apply<F, T>,
   mpl::identity<T> >::type type;
};
```

MPL

- Simplification
- Standard terminology
- Consistency
- Hides optimization and clever tricks inside the library

Troubles

Troubles

- Icky syntax
- Late type checking
- Multi megabyte error messages
- No real debugging support
- Long feedback cycle

What we might like to be able to do

Aims

- Some better syntax

What we might like to be able to do

Aims

- Some better syntax
- Type checking of our functions

What we might like to be able to do

Aims

- Some better syntax
- Type checking of our functions
- Comprehensible error messages

What we might like to be able to do

Aims

- Some better syntax
- Type checking of our functions
- Comprehensible error messages
- Interactive

What we might like to be able to do

Aims

- Some better syntax
- Type checking of our functions
- Comprehensible error messages
- Interactive
- Some way of running / debugging our code

Making life easier

Simplifications

- Reduced MPL concepts

Making life easier

Simplifications

- Reduced MPL concepts
- No iterators

Making life easier

Simplifications

- Reduced MPL concepts
- No iterators
- No arbitrary recursion

Making life easier

Simplifications

- Reduced MPL concepts
- No iterators
- No arbitrary recursion
- Other miscellaneous details

Some secondary aims

Usability

- The DSL should retain appropriate C++ syntax
- Generated code should be human readable
- Generated code should be good MPL practice

Implementation Language

Haskell

- Pure functional language
- Highly expressive type system
- Tight control over operator overloading
- Can be used interactively
- Multi platform

Outline

STL leads the way

STL Style ideas

- Containers
- Algorithms
- Iterators

Non STL extras

- Metafunctions
- Views

Fundamental types

```
mpl::int_<101>
mpl::bool_<true>
mpl::long_<202>
mpl::size_t<303>
mpl::integral_c<int, 303>

int, char, float, std::string, my_type ...
```

Sequence types

```
mpl::vector<int, char, ...>
mpl::list<float, std::string, ...>
mpl::deque<long, std::vector<int>, ...>

mpl::map<
  mpl::pair<int, char>,
  mpl::pair<float, std::string>,
  ...>
mpl::set<int, char, ...>
```

Metafunctions

```
template<typename T0, typename T1, ...>
struct my_metafunction
{
  typedef ... type;
};
```

Example metafunction

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

template<typename T>
struct remove_pointer<T*>
{
  typedef T type;
};
```

Our kind of metafunction

```
template<typename Seq, typename N>
struct at_even
{
  typedef typename mpl::at<
      Seq,
      mpl::times<mpl::int_<2>, N> >::type type;
};
```

Outline

Start in the usual place

```
template<typename N>
struct bump3
  : mpl::plus<
      N,
      mpl::int_<3> >
{};
```

Building blocks

- Variables
- Constants
- Applications of functions

DSL Core, first atttempt

```
data MPLExpr a = Const a |
                 Var String |
                 App String [MPLExpr a]
expr1 = 3
expr2 = x
expr3 = x + 3 * y
```

Generating code

The plan

- Convert constants to appropriate mpl types
- Variables placeholders for argument names
- For function applications, evaluate args and apply function

Generating code

The plan

- Convert constants to appropriate mpl types
- Variables placeholders for argument names
- For function applications, evaluate args and apply function

```
template<typename x, typename y>
struct fun {
  typedef typename mpl::plus<
    x, typename mpl::times<
      mpl::int_<3>,
      y>::type >::type type;
};
```

Generating better code

```
data MPLExpr a = Const a |
                 Var String |
                 App MPLFun [MPLExpr a]
```

Generating better code

```
data MPLExpr a = Const a |
                 Var String |
                 App MPLFun [MPLExpr a]
```

```
template<typename x, typename y>
struct fun {
  typedef mpl::plus<
    x, mpl::times<
     mpl::int_<3>,
     y> > type;
};
```

Logical operations

```
expr1 = a || b
expr2 = not a
expr3 = a && b
```

Logical operations

```
expr1 = a || b
expr2 = not a
expr3 = a && b

template<typename a, typename b, typename c>
struct fun {
  typedef mpl::or_<
   a, mpl::and_<
    b, c> > type;
};
```

Comparison operators

```
data MPLExpr a = Const a |
                 Var String |
                 App String [MPLExpr a]

expr1 = x < 303 --Broken
```

Trouble

Sub expressions are not necessarily of the same type as the resulting expression.

Existential Types

```
class Builtin a where
lit :: a -> String

data MPLSubExpr = forall a. Builtin a
  => MPLSubExpr { expr :: MPLExpr a }

 data MPLExpr a = Const a |
                  Var String |
                  App String [MPLSubExpr]
```

Our solution

- Encapsulates a type with a specified interface
- Think of good old C++ polymorphism

Now we can describe comparison operators

```
display_function "fun" (x < 3 || x > 5)
```

Now we can describe comparison operators

```
display_function "fun" (x < 3 || x > 5)

template <typename x>
struct fun
{
  typedef or_<
    less<
     x,
     mpl::int_<3> >,
    greater<
     x,
     mpl::int_<5> > > type;
};
```

Operators revisted

```
*DSL> let x = 101 :: Int
*DSL> :t x < 3 || x > 5
x < 3 || x > 5 :: Bool
*DSL> x < 3 || x > 5
True
```

How to debug?

- We'd like to run our expressions
- We'd like these runs to reflect the C++ semantics
- Provide a 2nd set of overloads with "Equivalent" semantics

Outline

Type traits

Issues

- We need type traits to do interesting stuff
- We're going to need a representation of types
- We'd still like some runtime behaviour

Type traits

Issues

- We need type traits to do interesting stuff
- We're going to need a representation of types
- We'd still like some runtime behaviour

```haskell
data CType = CType String |
             CPointer CType |
             CReference CType |
             CConst CType

my_type = CPointer $ CConst $ CType "std::string"
```

Type traits

```
*DSL> let x = Var "x" :: MPLExpr CType
*DSL> display_function "fun" $ add_pointer $ add_const x
template <typename x>
struct fun
{
  typedef typename add_pointer<
    typename add_const<
     x>::type>::type type;
};

*DSL> let x = CType "int"
*DSL> (add_pointer . add_const) x
int const *
```

Outline

Conditions

MPL conditional expression

- Eagerly evaluated mpl:: if_ function
- Lazily evaluated mpl:: eval_if function

Conditional outlines

```
template <typename b, typename t1, typename t2>
struct fun { // Returns t1 or t2
  typedef typename mpl::if_<
    b, t1, t2>::type type;
};

template <typename b, typename t1, typename t2>
struct fun { // Returns t1::type or t2::type
  typedef typename mpl::if_<
    b, t1, t2>::type type;
};
```

Eager evaluation is easy

```
display_function "fun" $ if_ b
  (add_pointer x)
  (add_reference x)
```

Eager evaluation is easy

```
display_function "fun" $ if_ b
  (add_pointer x)
  (add_reference x)

template <typename b, typename x>
struct fun {
  typedef typename if_<
    b,
    typename add_pointer<
     x>::type,
    typename add_reference<
     x>::type>::type type;
};
```

Lazy evaluation is not so easy

```
template <typename b, typename x>
struct fun {
  typedef typename eval_if<
    b,
    typename add_pointer<x>::type,
    typename add_reference<x>::type>::type type;
};
```

Lazy evaluation is not so easy

```
template <typename b, typename x>
struct fun {
  typedef typename eval_if<
    b,
    typename add_pointer<x>::type,
    typename add_reference<x>::type>::type type;
};

template <typename b, typename x>
struct fun {
  typedef typename eval_if<
    b,
    add_pointer<x>,
    add_reference<x> >::type type;
};
```

More expressive power in our DSL

```haskell
data MPLSubExpr = forall a. Builtin a =>
  MPLSubExpr { expr :: MPLExpr a , eval :: Bool }

data MPLExpr a = Var String |
                 Const a |
                 App MPLFun [MPLSubExpr]
```

A simple extension

Sub expressions are labelled to describe whether they should be evaluated eagerly or lazily.

Lazy conditionals now work

```
display_function "fun" $ eval_if b
  (add_pointer x)
  (add_reference x)
```

Lazy conditionals now work

```
display_function "fun" $ eval_if b
  (add_pointer x)
  (add_reference x)
```

```
template <typename b, typename x>
struct fun {
  typedef typename eval_if<
    b,
    add_pointer<
     x>,
    add_reference<
     x> >::type type;
};
```

Outline

Sequences

Where are we now?

- Typed constants, variables and function applications
- Support for eager and lazy evaluation
- A small code generator that produces expression from our core
- Overloads to provide runtime simulation of behaviour

Sequence examples

```
mpl::vector<int, char, std::string, my_type>
mpl::list<int, char, float>
mpl::deque<long, double, std::pair<int,int> >

mpl::map<
 mpl::pair<int, char>,
 mpl::pair<std::string, float> >
```

Sequence examples

```
mpl::vector<int, char, std::string, my_type>
mpl::list<int, char, float>
mpl::deque<long, double, std::pair<int,int> >

mpl::map<
 mpl::pair<int, char>,
 mpl::pair<std::string, float> >
```

Simplifications

- Multiple sequence concepts - we will support just one
- Associative sequences - also skipped for now
- Iterators - skip them as well!

Intrinsics

```
class Intrinsic a e n b | a -> e, a -> n, a -> b where
  front, back :: a -> e
  at :: a -> n -> e
  clear, pop_back, pop_front :: a -> a
  push_front, push_back :: a -> e -> a
  empty :: a -> b
  size :: a -> n
```

Intrinsics example

```
*DSL> display_function "fun" (front xs + back xs)
```

Intrinsics example

```
*DSL> display_function "fun" (front xs + back xs)

template <typename xs>
struct fun
{
  typedef plus<
    typename front<
     xs >::type ,
    typename back<
     xs >::type> type;
};
```

Runtime behaviour

```
*DSL> let xs = [1,2,3]
*DSL> front xs + back xs
4
```

Runtime

- Haskell lists are our proxy for sequences
- Many MPL sequence operations have Haskell analogs
- We'd need to be more intelligent when support multiple sequence concepts

Algorithms

Algorithm taxonomy

- Iteration algorithms
- Querying algorithms
- Transformation algorithms

```
typename mpl::fold<xs, init, f>::type
typename mpl::contains<xs, x>::type
typename mpl::reverse<xs>::type
```

Views

```
mpl::empty_sequence<>
mpl::filter_view<xs, f>
mpl::joint_view<xs, ys>
mpl::single_view<x>
mpl::transform<xs, f>
mpl::zip_view<mpl::vector<xs, ys, zs> >
```

Easy stuff

Easy algorithms

- remove
- count
- contains
- reverse

Easy views

- joint_view
- single_view
- empty_view

Neutral constructors

```
*DSL> let xs = [1,2,3]
*DSL> :t joint_view xs (vector [1,2,3])
joint_view xs (vector [4,5,6]) :: [Integer]
*DSL> joint_view xs (vector [4,5,6])
[1,2,3,4,5,6]

*DSL> let xs = Var "xs" :: MPLExpr [Int]
*DSL> :t joint_view xs (vector [1,2,3])
joint_view xs (vector [1,2,3]) :: MPLExpr [Int]
*DSL> joint_view xs (vector [1,2,3])
App (MPLFun {name = "joint_view", requiresType = False})
```

The (currently) partially impossible

```
zip :: [[?]] -> [[?]]
```

Typing discipline

Our type system is too strict/rigid to permit zip in its current form. Consider zipping a sequence of int values with a list of types. Neither the input or output types of zip can be described in our type system.

The not so easy

```
typename mpl::transform<xs, f>::type
```

A new feature

transform and many other interesting sequence algorithms are parameterized by one or more functions. We actually already support passing functions as arguments, but it's not very easy to get your hands on them.

MPL Lambdas

```
typename mpl::transform<
  xs,
  add_pointer<_> >::type

typename mpl::transform<
  xs,
  mpl::plus<_, mpl::int_<101> > >::type
```

An example of the problem

Good

```
display_function "fun" (add_pointer x)
```

An example of the problem

Good

```
display_function "fun" (add_pointer x)
```

Bad

```
display_function "fun" (transform xs add_pointer)
```

- add_pointer if of type MPLExpr CType −> MPLExpr CType
- We'd like it to be of type MPLExpr XXX

A 1st plan for passing functions

Functions

- For our runtime behaviour we just represent functions as Haskell functions of type a−>b
- For our compile time representations, we need objects describing the type of the function, and also carrying information about it's name etc.

A function object constant for each function

```
remove_if xs is_pointer_f :: MPLExpr [CType]
transform2 xs ys plus :: MPLExpr [Int]
transform2 xs ys less_equal :: MPLExpr [Bool]
```

A simple approach

Add constants that represent each function:

- Each trait
- Each unary and binary operator

Discussion of plan A

Plan A

- Convenient to use for simple cases
- Lacks flexibility
- Inconvenient to code - loads of repetition
- Didn't polute the DSL core

Forming a plan B

Some potential improvements

- Automatically convert DSL functions to "function objects"

Forming a plan B

Some potential improvements

- Automatically convert DSL functions to “function objects”
- More flexibility - binding of arguments

Forming a plan B

Some potential improvements

- Automatically convert DSL functions to "function objects"
- More flexibility - binding of arguments
- Something like our old friends bind1st and bind2nd

Forming a plan B

Some potential improvements

- Automatically convert DSL functions to “function objects”
- More flexibility - binding of arguments
- Something like our old friends bind1st and bind2nd

The new facilities

- bind1st
- bind2nd
- bind1
- bind2

A bit more explanation

```
is_pointer :: MPLExpr CType -> MPLExpr Bool
bind1 is_pointer :: MPLExpr (Fun (CType -> Bool))

(<) :: MPLExpr Int -> MPLExpr Int -> MPLExpr Int
bind2 (<) :: MPLExpr (Fun (Int -> Int -> Bool))
bind1st (<) x :: MPLExpr (Fun (Int -> Bool))
```

Discussion of Plan B

Binders

- More general than the "convenient constants" approach
- Actually required some polution to the core DSL
- Nowhere near as effective as MPL Lambda syntax

Outline

Extension on the cheap

Basic extension

- Refer to functions not defined in the DSL directly
- No runtime debugging support

```
*DSL> let f = Const $ Fun "my_function"
        :: MPLExpr (Fun (Int -> Bool))
*DSL> display_function "fun" $ remove_if xs f
template <typename xs>
struct fun
{
  typedef typename remove_if<
    xs,
    my_function >::type type;
};
```

Genuine extensions

The standard routine

- Add a Haskell Type class to permit overload of the new function
- Add a code generation overload
- Add a runtime behaviour overload
- Optionally add constants so it can be passed as an argument

Extension example

```
class Mirrorable a where
  mirror :: a -> a

instance (Builtin a) => Mirrorable (MPLExpr [a]) where
  mirror = App (MPLFun "mirror" True)
    [MPLSubExpr xs True]

instance Mirrorable [a] where
  mirror xs = xs ++ reverse xs
```

Future directions

Ideas

- Improved lambda strategy
- Recursion support
- Easier reference to previously defined functions
- More accurate modelling of MPL concepts - iterators, diffent sequences etc.
- Broader MPL and Boost.TypeTraits support
- Ideally both runtime and compile time behaviour from a single representation

Credits and thanks

Credits

- Oliver Mueller
- Lennart Augustsson's blog "things that amuse me", an online DSL goldmine
- Joel, Hartmut and the Spirit community
- Aleksey Gurtovoy and David Abrahams for the MPL