Building finite-element matrix expressions with Boost Proto and the Eigen library

Bart Janssens

17 May 2013

Outline

Context

- Navier-Stokes solver for incompressible flow
- To be combined with a model for particle dispersion
- Finite Element Method discretization
- PhD work promoted by:
  - T. Arts (von Karman Institute for Fluid Dynamics, BEL)
  - W. Bosschaerts (Royal Military Academy, BEL)
  - K. Limam (La Rochelle University, FRA)
- Implemented in the Coolfluid collaborative platform (LGPL): http://coolfluid.github.com

Context

- Finite Element Method: Solve equations from physics
- Most algorithm steps are generic, specifics depend on the physical problem
  - We use a domain specific language to describe the physics
  - The back end is reused
  - Boilerplate code is avoided
- Simple example: Heat conduction

$$A_e = \int_{\Omega_i} k \nabla \mathbf{N_T}^{\mathrm{T}} \nabla \mathbf{N_T} \, \mathrm{d}\Omega_i$$

```
element_quadrature
(
  _A(T,T) += k * transpose(nabla(T)) * nabla(T)
)
```

Context

- Mathematics behind Finite Element Method: small matrices
- We use the Eigen library
  - Matrix dimensions set at compile time
  - Many optimizations
  - Fastest we could find
  - Uses expression templates
- Our Domain Specific Language
  - Implemented using Proto
  - Uses Eigen for matrix mathematics
  - Can be extended with “user-defined terminals”

```
element_quadrature
(
  _A(T,T) += k * transpose(nabla(T)) * nabla(T)
)
```

Today's goals

1. Show how to use Eigen inside a higher-level Proto language
   - One expression template nested into another expression template library
   - $\Rightarrow$ Problems with dangling references
   - Can be fixed using Proto grammars
2. Add user-defined terminals
   - Easily extend the provided language without needing to touch the grammar
   - Purpose: Specialized optimal code, missing functionality, code reuse, ...
3. Link with FEM problem
   - Show how to link Proto with typical run time data structures

Work method

- Proto tutorials:
  1. Expressions and transforms basics
  2. Expose the problem when combining with Eigen
  3. Proposed solution for Eigen
  4. Introduce user-defined terminals
  5. Make some links to FEM-like data
- Stand-alone example code: http://github.com/barche/eigen-proto
- Depends only on Eigen and Proto

What is Proto?

Tool to build Embedded Domain Specific Languages:

- Build expression trees
- Assign meaning to the expressions
- Build expression templates without the hassle
- Supplies its own language for describing a language

Main components:

- Expressions
- Grammars
- Transforms

Terminals

Simplest Proto expression: the “terminal”:

```
int i = 1; // Create an int
proto::literal<int&> int_term(i); // Explicit proto terminal
proto::lit(i); // In-place construction
proto::display_expr(proto::lit(i)); // Print it
```

Output: `terminal(1)`

- Terminal is a Proto object that holds a reference to an integer here
- We can get the value using `proto::value`
- We can combine terminals into more complicated expressions

Expressions

Let's show some expressions:

```
proto::display_expr(int_term * 2 + 3); // Arithmetic
proto::display_expr(int_term = 2); // Assignment
```

This prints:

```
plus(
    multiplies(
        terminal(3)
      , terminal(2)
    )
  , terminal(3)
)
```

```
assign(
    terminal(3)
  , terminal(2)
)
```

Expressions

Let's show some expressions:

```
proto::display_expr(int_term(1,2)); // Function call
proto::display_expr(int_term << "shift" << "things"); // shift
```

This prints:

```
function(
    terminal(3)
  , terminal(1)
  , terminal(2)
)
```

```
shift_left(
    shift_left(
        terminal(3)
      , terminal(shift)
    )
  , terminal(things)
)
```

Grammars

Grammars indicate what rules an expression should follow, i.e. if we only want to add up integers:

```
struct add_ints_grammar :
  proto::plus< proto::terminal<int>, proto::terminal<int> >
{
};
```

- `add_ints_grammar` is a Proto grammar
- Note that it's declared as an empty struct
- All the magic is in the type we inherit from
- This is Proto's way of describing grammars!

Checking expressions

- Grammars can be used to check expressions:

```
template<typename ExprT>
void check_expr(const ExprT&)
{
  BOOST_MPL_ASSERT((
    proto::matches<ExprT, add_ints_grammar>
  ));
}
// Adding two ints is OK
check_expr(proto::lit(i) + 2);
// Anything else won't compile:
check_expr(proto::lit(i) + 2u);
```

- Compile error if anything is wrong
- Still doesn't actually do anything

- Transforms allow grammars to do something useful
- Let's first build a functor:

```
struct evaluate_plus : proto::callable
{
  typedef int result_type;
  int operator()(int a, int b) const { return a + b; }
};
```

- And then use it as a transform:

```
struct add_ints_transform :
  proto::when<
    add_ints_grammar,
    evaluate_plus(proto::_value(proto::_left),
                  proto::_value(proto::_right))>
{};
```

- Transforms are used as a functor on an expression:

```
const int result = add_ints_transform()(lit(1) + lit(2));
```

Add some more calculations...

- Simplify using recursion and the default transform:

```
struct calculator_transform :
 or_
 <
  // Replace terminals with their value:
  when< terminal<_>, _value >,
  when<or_ // When we have +, - * or / ...
  <
   plus<calculator_transform, calculator_transform>,
   minus<calculator_transform, calculator_transform>,
   multiplies<calculator_transform, calculator_transform>,
   divides<calculator_transform, calculator_transform>
  >,
  _default<calculator_transform> > // ...Do C++ default
 >
{};
```

```
int a = 2; double b = 0.5; unsigned int c = 3;
calculator_transform()( (lit(a) + lit(c)) * lit(b) );
```

What we have so far...

- A transform that can evaluate expressions using +, -, * and /
- Expressions can be arbitrarily complex because of recursion in the transform
- Operands can have any type because of the wildcard
- Should work, as long as the operators are defined on the types that we use
- Let's find out what happens with Eigen...

Eigen test

- Construct $A = \begin{bmatrix} 2 & 2 \end{bmatrix}$, $B = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$ and $C = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

```
// Typedefs for matrix types
typedef Eigen::Matrix<double, 1, 2> AT;
typedef Eigen::Matrix<double, 2, 1> BT;
typedef Eigen::Matrix<double, 2, 2> CT;

// Construct the matrices
AT a_mat; a_mat.setConstant(2);
BT b_mat; b_mat.setConstant(2);
CT c_mat; c_mat.setConstant(1);

// Build proto terminals
proto::literal<AT&> a(a_mat);
proto::literal<BT&> b(b_mat);
proto::literal<CT&> c(c_mat);
```

Eigen test

- Calculate BA, expected to yield $\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$

```
calculator_transform eval;
std::cout << eval(b*a) << std::endl;
```

Eigen test

- Calculate BA, expected to yield $\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$

```
calculator_transform eval;
std::cout << eval(b*a) << std::endl;
```

- Works!

Eigen test

- Calculate BA, expected to yield $\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$

```
calculator_transform eval;
std::cout << eval(b*a) << std::endl;
```

- Works!
- Calculate $(BA)C$, expected to yield $\begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}$

```
std::cout << eval((b*a)*c) << std::endl;
```

Eigen test

- Calculate BA, expected to yield $\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$

```
calculator_transform eval;
std::cout << eval(b*a) << std::endl;
```

- Works!
- Calculate $(BA)C$, expected to yield $\begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}$

```
std::cout << eval((b*a)*c) << std::endl;
```

- Fails! (crash or bad result or correct result)

What went wrong?

- Eigen: expression templates ⇒ Return types?

```
// Result of calculator_transform()(expr)
template<typename ExprT>
void print_result_type(const ExprT& expr)
{
  boost::result_of<calculator_transform(ExprT)>::type::
      print_error();
}
```

- A*B:

```
CoeffBasedProduct<const Matrix<double, 2, 1, 0, 2, 1> &,
    const Matrix<double, 1, 2, 1, 1, 2> &, 256>
```

- (A*B)*C: Spot the reference

```
CoeffBasedProduct<const CoeffBasedProduct<const Matrix<
    double, 2, 1, 0, 2, 1> &, const Matrix<double, 1, 2, 1,
     1, 2> &, 256> &, const Matrix<double, 2, 2, 0, 2, 2>
    &, 6>
```

What went wrong?

- Similar effect using only 4x4 matrices
- A*B:

```
Eigen::CoeffBasedProduct<const Eigen::Matrix<double, 4, 4,
    0, 4, 4> &, const Eigen::Matrix<double, 4, 4, 0, 4, 4>
    &, 6>
```

- (A*B)*C: Reference to temporary matrix

```
Eigen::CoeffBasedProduct<const Eigen::Matrix<double, 4, 4,
    0, 4, 4> &, const Eigen::Matrix<double, 4, 4, 0, 4, 4>
    &, 6>
```

- For performance reasons, Eigen may create temporary matrices and use references to them

Expression templates and temporaries

- This works with Eigen matrices:

```
(b_mat*a_mat)*c_mat; // Eigen: Single C++ statement
```

```
proto::display_expr((b*a)*c); // The proto equivalent
```

```
multiplies(
```

```
    multiplies(
        terminal(2
                 2)
      , terminal(2 2)
    )
```

```
  , terminal(1 1
             1 1)
)
```

- No longer a single statement!

How can this be solved?

- Problem: Temporaries disappear when Eigen still holds references to them
- Solution: Make sure the temporaries don't disappear!
- How?
  1. Identify problematic expressions
  2. Transform problematic expressions
  3. Reserve storage for the temporary
  4. Account for the change in product evaluation
  5. Transform the entire expression tree
- Proto makes this easier than it might seem!

Identify problematic expressions

We just consider the multiplies expression as a problem:

```
struct wrap_expression :
  proto::or_
  <

    proto::when
    <
      proto::multiplies<proto::_, proto::_>,




    >,

  >
{
};
```

Transform problematic expressions

Work done by `do_wrap_expression`:

```
struct wrap_expression :
  proto::or_
  <

    proto::when
    <
      proto::multiplies<proto::_, proto::_>,
      do_wrap_expression(proto::functional::make_multiplies
      (
        wrap_expression(proto::_left), wrap_expression(proto::
            _right)
      ))
    >,

  >
{
};
```

Transform problematic expressions

Create a `stored_result_expression` to hold the result

```
struct do_wrap_expression : proto::transform<do_wrap_expression>
{
  template<typename ExprT, typename StateT, typename DataT>
  struct impl : proto::transform_impl<ExprT, StateT, DataT>
  {
    // ... Some ugly result type calculation
    typedef stored_result_expression<expr_val_type, value_type>
        result_type;

    result_type operator()(typename impl::expr_param expr,
        typename impl::state_param state, typename impl::
        data_param data)
    {
      return result_type(expr);
    }
  };
}
```

Reserve storage for the temporary

Just add a value member to the expression:

```
template<typename ExprT, typename ValueT>
struct stored_result_expression :
 proto::extends<ExprT, stored_result_expression<ExprT, ValueT> >
{
  EIGEN_MAKE_ALIGNED_OPERATOR_NEW

  typedef proto::extends< ExprT, stored_result_expression<ExprT,
       ValueT> > base_type;

  explicit stored_result_expression(ExprT const &expr = ExprT())
    : base_type(expr)
  {
  }

  /// Temporary storage for the result of the expression
  mutable ValueT value;
};
```

Calculation of ValueT was in the skipped ugly part...

Storage ValueT calculation

Get the result of evaluating the product:

```
typedef typename result_of<calculator_transform(ExprT, StateT,
    DataT)>::type result_ref_type;

typedef typename remove_reference<result_ref_type>::type
    value_type;

typedef typename remove_const<typename remove_reference<ExprT>::
    type>::type expr_val_type;

typedef stored_result_expression<expr_val_type, value_type>
    result_type;
```

So the `stored_result_expression` ValueT is actually provided by `calculator_transform`

Account for the change in product evaluation

Original transform:

```
struct calculator_transform :
  or_
  <
    when< terminal<_>, _value >, // Replace terminals with their
         value
    when<or_ // When we have +, - * or / ...
    <
      plus<calculator_transform, calculator_transform>,
      minus<calculator_transform, calculator_transform>,
      multiplies<calculator_transform, calculator_transform>,
      divides<calculator_transform, calculator_transform>
    >,
    _default<calculator_transform> > // ... Do what C++ would do
  >
{
};
```

Account for the change in product evaluation

```
struct calculator_transform :
  or_
  <
    when< terminal<_>, _value >,
    when // Multiplies special treatment
    <
       multiplies< calculator_transform, calculator_transform >,
       do_eigen_multiply(_, calculator_transform(_left),
            calculator_transform(_right))
    >,
    when<or_ // When we have +, - or / ...
    <
      plus<calculator_transform, calculator_transform>,
      minus<calculator_transform, calculator_transform>,
      divides<calculator_transform, calculator_transform>
    >,
    _default<calculator_transform> > // ... Do what C++ would do
  >
{};
```

Account for the change in product evaluation

```
struct do_eigen_multiply : proto::callable
{
  template<typename Signature> struct result;

  template<class ThisT, class ExprT, class LeftT, class RightT>
  struct result<ThisT(ExprT, LeftT, RightT)>
  {
    typedef typename product_value_type<LeftT, RightT>::type&
        type;
  };

  template<typename ExprT, typename LeftT, typename RightT>
  typename product_value_type<LeftT, RightT>::type& operator()(
      ExprT& expr, const LeftT& left, const RightT& right)
      const
  {
    expr.value = left*right;
    return expr.value;
  }
};
```

Actual ValueT computation

Uses Eigen-specific code:

```
template<typename LeftT, typename RightT>
struct product_value_type
{
  // Remove references
  typedef typename remove_reference<LeftT>::type left_unref;
  typedef typename remove_reference<RightT>::type right_unref;
  // The type returned by left*right (expression template)
  typedef typename Eigen::ProductReturnType<left_unref,
      right_unref>::Type product_type;
  // The storable matrix that left*right corresponds to
  typedef typename Eigen::MatrixBase<product_type>::PlainObject
      type;
};
```

Incomplete in the presence of scalars!

Completing the transformation

```
struct wrap_expression :
  proto::or_
  <
    proto::terminal<proto::_>,
    proto::when
    <
      proto::multiplies<proto::_, proto::_>,
      do_wrap_expression(proto::functional::make_multiplies
      (
        wrap_expression(proto::_left), wrap_expression(proto::
            _right)
      ))
    >,
    proto::nary_expr< proto::_, proto::vararg<wrap_expression> >
  >
{};
```

Let's test!

- Previously failing case:

```
wrap_expression wrap;
calculator_transform eval;
std::cout << eval(wrap((b*a)*c));
std::cout << eval(wrap((b*a)*c*(b*a)+c*b*a-(c+c)*b*a));
```

Let's test!

- Previously failing case:

```
wrap_expression wrap;
calculator_transform eval;
std::cout << eval(wrap((b*a)*c));
std::cout << eval(wrap((b*a)*c*(b*a)+c*b*a-(c+c)*b*a));
```

- Works!
- Let's recap what happens:
  1. `wrap_expression` wraps all multiplies in the tree
  2. During wrapping, `calculator_transform` is asked for the result type
  3. `calculator_transform` then evaluates the wrapped expression tree, and assumes all multiplies expressions have a correctly typed value member

User-defined terminals

- Goal: Allow users to extend the language without touching the grammar or transforms
- Use cases:
  - Add new functionality
  - Optimized implementation
  - Code reuse
- Current scope: Easily add more Eigen functionality
  - transpose
  - diagonal
  - column
  - ...

Proto terminals reminder

- Consider two structs:

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

struct my_callable {};
```

- This is a terminal:

```
terminal< user_op<my_callable> >::type const my_op = {};
```

- Expressions like this are perfectly fine:

```
my_op(1,2,3);
```

- and matched by this grammar:

```
function< terminal< user_op<_> >, vararg<_> >
```

Outline for user-defined terminals

1. Grammar catches all `user_op` terminals
2. Special transform handles calling `my_callable` on the arguments
3. User creates a functor `my_callable`
4. User creates a terminal of type
   `terminal< user_op<my_callable> >`
5. Terminal is used as function call in expressions

Catch user ops

```
struct calculator_transform :
  or_
  <
    // Add new transform here
    when< terminal<_>, _value >,
    when // Multiplies special treatment
    <
       multiplies< calculator_transform, calculator_transform >,
       do_eigen_multiply(_, calculator_transform(_left),
           calculator_transform(_right))
    >,
    when<or_ // When we have +, - or / ...
    <
      plus<calculator_transform, calculator_transform>,
      minus<calculator_transform, calculator_transform>,
      divides<calculator_transform, calculator_transform>
    >,
    _default<calculator_transform> > // ... Do what C++ would do
  >
{};
```

Catch user ops

- We add this new grammar and transform:

```
when
<
  function< terminal< user_op<_> >, vararg<_> >,
  evaluate_user_op(function< _, vararg<calculator_transform
      > >)
>
```

- Evaluate any `user_op` terminal used as a function on any number of arguments
- Arguments are evaluated using the `calculator_transform` itself
- `evaluate_user_op` is a primitive transform

Call the user op

```
struct evaluate_user_op : proto::transform< evaluate_user_op >
{
  template<typename ExprT, typename StateT, typename DataT>
  struct impl : proto::transform_impl<ExprT, StateT, DataT>
  {
    // Calculate the type of the functor that the user supplied
        ... (see later)

    // Compute its result type
    static const int nb_args = proto::arity_of<ExprT>::value-1;
    typedef typename compute_result_type<nb_args>::type
        result_type;

    result_type operator()(typename impl::expr_param expr)
    {
      return dispatch(mpl::int_<nb_args>(), expr);
    }
  };
};
```

Getting the functor type

```
// 0-th child is the terminal
typedef typename result_of<proto::_child0(ExprT)>::type
    callable_term_t;
// Its value is returned by reference
typedef typename result_of<proto::_value(callable_term_t)>::type
     callable_ref_t;
// And finally we have what we want
typedef typename remove_reference<callable_ref_t>::type::
    callable_t callable_t;
```

Calculating the result type

```
// Helper struct to calculate result types
template<int NbArgs, int dummy = 0>
struct compute_result_type;

template<int dummy> // 1 argument
struct compute_result_type<1, dummy>
{
  typedef typename result_of<_child1(ExprT)>::type child1_t;
  typedef typename result_of<callable_t(child1_t)>::type type;
};

template<int dummy> // 2 arguments
struct compute_result_type<2, dummy>
{
  typedef typename result_of<_child1(ExprT)>::type child1_t;
  typedef typename result_of<_child2(ExprT)>::type child2_t;
  typedef typename result_of<callable_t(child1_t, child2_t)>::
      type type;
};
```

Dispatch on the number of arguments

```
result_type dispatch(mpl::int_<1>, typename impl::expr_param
    expr)
{
  return callable_t()(proto::child_c<1>(expr));
}

result_type dispatch(mpl::int_<2>, typename impl::expr_param
    expr)
{
  return callable_t()(proto::child_c<1>(expr), proto::child_c
      <1>(expr));
}
```

A user defined terminal

```
struct do_transpose
{
  template<typename Signature> struct result;

  template<class ThisT, typename MatrixT>
  struct result<ThisT(MatrixT)>
  {
    typedef Eigen::Transpose<MatrixT> type;
  };

  template<typename MatrixT>
  Eigen::Transpose<MatrixT> operator()(MatrixT& mat)
  {
    return mat.transpose();
  }
};

terminal< user_op<do_transpose> >::type const transpose = {};
```

Let's test

- Consider $c = \begin{bmatrix} 5 & 1 \\ 5 & 1 \end{bmatrix}$

```
eval(wrap(transpose(c)))
```

- this yields: $c = \begin{bmatrix} 5 & 5 \\ 1 & 1 \end{bmatrix}$
- What about this:

```
eval(wrap(transpose(transpose(c))))
```

- Should work because of recursion:

```
when< function< terminal< user_op<_> >, vararg<_> >,
  evaluate_user_op(function<_, vararg<calculator_transform>
        >)>
```

- Fails!

What’s wrong?

```
struct do_transpose
{
  template<typename Signature> struct result;

  template<class ThisT, typename MatrixT>
  struct result<ThisT(MatrixT)>
  {
    typedef Eigen::Transpose<MatrixT> type;
  };

  template<typename MatrixT>
  Eigen::Transpose<MatrixT> operator()(MatrixT& mat)
  {
    return mat.transpose();
  }
};
```

Reference to MatrixT won’t work for an Eigen::Transpose

Solution: two overloads

```
template<typename MatrixT>
Eigen::Transpose<MatrixT> operator()(MatrixT mat)
{
  return mat.transpose();
}

template<int Rows, int Cols>
Eigen::Transpose< Eigen::Matrix<double, Rows, Cols> > operator()
    (Eigen::Matrix<double, Rows, Cols>& mat)
{
  return mat.transpose();
}
```

- If it's a matrix, pass by reference
- If it's something else, pass by value
- Works:

```
eval(wrap(transpose(transpose(transpose(c)))))
```

Returning a matrix

Constant matrix filled with 2:

```
struct make_twos
{
  template<typename Signature> struct result;

  template<class ThisT, typename MatrixT>
  struct result<ThisT(MatrixT)>
  {
    typedef MatrixT type;
  };

  template<typename MatrixT>
  MatrixT operator()(const MatrixT& mat)
  {
    return MatrixT::Constant(2.);
  }
};

proto::terminal< user_op<make_twos> >::type const twos = {};
```

Some tests

```
eval(wrap(twos(c)));
eval(wrap(twos(b)*twos(a)));
eval(wrap((twos(c)+twos(c))*c)); // oops
```

- Last one fails
- Dangling references again!
- Solution: Don't return a temporary, but a reference

```
template<typename MatrixT>
MatrixT operator()(const MatrixT& mat)
{
  return MatrixT::Constant(2.);
}
```

- How?

Returning a matrix

```
struct make_twos
{
  template<typename Signature>
  struct result;

  template<class ThisT, typename MatrixT>
  struct result<ThisT(MatrixT)>
  {
    typedef MatrixT type;
  };

  template<typename StoredT, typename MatrixT>
  StoredT& operator()(StoredT& stored, const MatrixT&)
  {
    stored.setConstant(2.);
    return stored;
  }
};
```

Returning a matrix

Required changes:

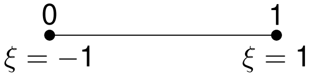
- Add user ops to the wrapping grammar
- Store when returning a reference
- Add stored argument to the user op call
- Problem: `result_of` doesn't match actual call

Recap

- Use Eigen matrices inside Proto
- 1-to-1 mapping between Eigen and Proto expression
- Completely pointless!
- Real use case:
  - Collections of matrices
  - Generated matrices
  - Operations on FEM data structure
- Next: Calculation of element centroids using a simple FEM-based system

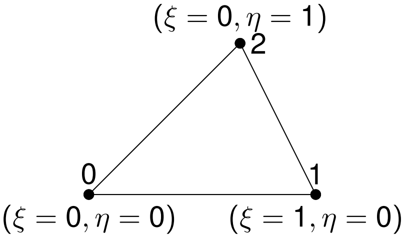
Finite Elements

1D Lines:



$$N = \left[\tfrac{1}{2}(1-\xi) \quad \tfrac{1}{2}(1+\xi)\right]$$

2D Triangles:



$$N = \left[\xi \quad \eta \quad 1-\xi-\eta\right]$$

Line element

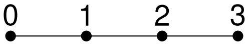
```
struct line1d
{
  static const int nb_nodes = 2;
  static const int dimension = 1;
  typedef Eigen::Matrix<double, 1, dimension> coord_t;
  typedef Eigen::Matrix<double, 1, nb_nodes> shape_func_t;

  static shape_func_t shape_function(const coord_t& c)
  {
    const double xi = c[0];
    shape_func_t result;
    result[0] = 0.5*(1.-xi); result[1] = 0.5*(1.+xi);
    return result;
  }

  static const coord_t& centroid()
  {
    static const coord_t c = coord_t::Constant(0.);
    return c;
  }
};
```

Triangle element

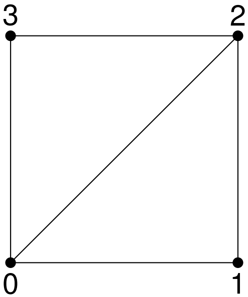
```
struct triag2d
{
  static const int nb_nodes = 3;
  static const int dimension = 2;
  typedef Eigen::Matrix<double, 1, dimension> coord_t;
  typedef Eigen::Matrix<double, 1, nb_nodes> shape_func_t;

  static shape_func_t shape_function(const coord_t& c)
  {
    const double xi = c[0]; const double eta = c[1];
    shape_func_t result;
    result[0] = xi; result[1] = eta; result[2] = 1. - xi - eta;
    return result;
  }

  static const coord_t& centroid()
  {
    static const coord_t c = coord_t::Constant(1./3.);
    return c;
  }
};
```

Simple unstructured mesh

Lines:



Triangles:



Unstructured mesh code

```
struct mesh_data
{
  mesh_data(const int nb_nodes, const int nb_elems, const int
        nb_nodes_per_elem, const int dimension) :
    coordinates(boost::extents[nb_nodes][dimension]),
    connectivity(boost::extents[nb_elems][nb_nodes_per_elem])
  {
  }

  typedef boost::multi_array<double, 2> coordinates_t;
  typedef boost::multi_array<int, 2> connectivity_t;
  coordinates_t coordinates;
  connectivity_t connectivity;
};
```

- Element type not stated
- No compile-time size info!

Steps for activating Proto

To compute the centroids using proto

1. Define helper data with required compile-time information
2. Define terminals and functions for the element coordinates, the centroid mapped coordinates and the shape function matrix
3. Write the grammar and transforms
4. Create the data
5. Write the proper expression and execute it

Proto helper data: templated!

```
template<typename ElementT> struct dsl_data
{
  typedef ElementT element_t;
  typedef Eigen::Matrix<double, element_t::nb_nodes, element_t::
      dimension> coord_mat_t;

  dsl_data(const fem::mesh_data& d) : mesh_data(d) {}

  void set_element(const int e)
  {
    for(int i = 0; i != element_t::nb_nodes; ++i)
      for(int j = 0; j != element_t::dimension; ++j)
        coord_mat(i,j) = mesh_data.coordinates[mesh_data.
            connectivity[e][i]][j];
  }

  const fem::mesh_data& mesh_data;
  coord_mat_t coord_mat;
  typename element_t::shape_func_t shape_func;
};
```

Element coordinate matrix

```
struct element_coords_tag {};

struct eval_element_coord : proto::callable
{
  template<typename DataT>
  const typename DataT::coord_mat_t& operator()(const DataT&
        data) const
  {
    return data.coord_mat;
  }
};

/// Return the element coordinates matrix
proto::terminal< element_coords_tag >::type const element_coords
      = {};
```

Element centroid mapped coordinate

```
struct centroid_tag {};

struct eval_centroid : proto::callable
{
  template<typename DataT>
  const typename DataT::element_t::coord_t& operator()(const
      DataT&) const
  {
    return DataT::element_t::centroid();
  }
};

/// Return the mapped coordinate referring to the centroid
proto::terminal< centroid_tag >::type const centroid = {};
```

Shape function value

```
struct shape_func_tag {};

struct eval_shape_func : proto::callable
{
  template<typename CoordT, typename DataT>
  const typename DataT::element_t::shape_func_t& operator()(
      const CoordT& coord, DataT& data) const
  {
    data.shape_func = DataT::element_t::shape_function(coord);
    return data.shape_func;
  }
};

/// Return the shape function
proto::terminal< shape_func_tag >::type const N = {};
```

The grammar

```
struct fem_grammar :
or_
<
  when<terminal<element_coords_tag>, eval_element_coord(_data)>,
  when<terminal<centroid_tag>, eval_centroid(_data)>,
  when
  <
    function<terminal<shape_func_tag>, _>,
    eval_shape_func(fem_grammar(_child1), _data)
  >,
  _default<fem_grammar>
>
{
};
```

Getting compile-time information

We use a helper function to run an expression:

```
template<typename ExprT>
void for_each_element(const fem::mesh_data& mesh, const ExprT&
    expr)
{
  // Allowed element types
  typedef mpl::vector2<fem::line1d, fem::triag2d> element_types;
  mpl::for_each<element_types>(element_looper<ExprT>(mesh, expr)
      );
}
```

Getting compile-time information

The MPL functor knows the concrete element type:

```
template < typename ElemT >
void operator()(ElemT) const
{
  if(ElemT::dimension != mesh.coordinates.shape()[1] ||
     ElemT::nb_nodes != mesh.connectivity.shape()[1])
     return;

  dsl_data<ElemT> data(mesh);
  const int nb_elems = mesh.connectivity.size();
  for(int i = 0; i != nb_elems; ++i)
  {
    data.set_element(i);
    fem_grammar()(expr, 0, data);
  }
}
```

Running the expressions

```
mesh_data line_mesh(4, 3, 2, 1);
// Mesh filling went here
std::cout << "line mesh centroids:" << std::endl;
for_each_element(line_mesh,
  cout_ << N(centroid)*element_coords << "\n");

mesh_data triag_mesh(4, 2, 3, 2);
// Mesh filling went here
std::cout << "triag mesh centroids:" << std::endl;
for_each_element(triag_mesh,
  cout_ << N(centroid)*element_coords << "\n");
```

Output:

```
line mesh centroids:
0.5
1.5
2.5
triag mesh centroids:
0.666667 0.333333
0.333333 0.666667
```

If we go wrong...

```
for_each_element(triag_mesh,
  cout_ << centroid*element_coords << "\n");
```

Deep in the template backtrace:

```
/usr/local/include/eigen3/Eigen/src/Core/util/StaticAssert.h
    :183:5: note: expanded from macro '
    EIGEN_STATIC_ASSERT_SAME_MATRIX_SIZE'
    YOU_MIXED_MATRICES_OF_DIFFERENT_SIZES)
    ^
/usr/local/include/eigen3/Eigen/src/Core/../plugins/
    MatrixCwiseBinaryOps.h:24:10: note: in instantiation of
    member function 'Eigen::CwiseBinaryOp<Eigen::internal::
    scalar_product_op<double, double>, const Eigen::Transpose<
    const Eigen::Matrix<double, 1, 1, 0, 1, 1> >, const Eigen::
    Matrix<double, 2, 1, 0, 2, 1> >::CwiseBinaryOp' requested
    here
  return EIGEN_CWISE_PRODUCT_RETURN_TYPE(Derived,OtherDerived)(
      derived(), other.derived());
```

A complicated example

```cpp
m_assembly->add_component(create_proto_action
(
  action_name,
  elements_expression
  (
    AllElementsT(),
    group
    (
      _A = _0, _T = _0,
      for_generic_elements
      (
        compute_tau(u, nu_eff, u_ref, lit(tau_ps), lit(tau_su), lit(tau_bulk)),
        element_quadrature
        (
          _A(p    , u[_i]) += transpose(N(p) + tau_ps*u_adv*nabla(p)*0.5) * nabla(u)[_i] + tau_ps * transpose(nabla(p)[_i]) * u_adv*nabla(u),
          _A(p    , p)     += tau_ps * transpose(nabla(p)) * nabla(p) / rho,
          _A(u[_i], u[_i]) += nu_eff * transpose(nabla(u)) * nabla(u) + transpose(N(u) + tau_su*u_adv*nabla(u)) * u_adv*nabla(u),
          _A(u[_i], p)     += transpose(N(u) + tau_su*u_adv*nabla(u)) * nabla(p)[_i] / rho,
          _A(u[_i], u[_j]) += transpose((tau_bulk + 0.33333333333333*nu_eff)*nabla(u)[_i]
                              + 0.5*u_adv[_i]*(N(u) + tau_su*u_adv*nabla(u))) * nabla(u)[_j],
          _T(p    , u[_i]) += tau_ps * transpose(nabla(p)[_i]) * N(u),
          _T(u[_i], u[_i]) += transpose(N(u) + tau_su*u_adv*nabla(u)) * N(u)
        )
      ),
      for_specialized_elements(supg_specialized(p, u, u_adv, nu_eff, u_ref, rho, _A, _T)),
      system_rhs += -_A * _x,
      _A(p) = _A(p) / theta,
      system_matrix += invdt() * _T + theta * _A
    )
  )
));
```

Conclusion

- Nesting of expression template libraries
- Definition of user-defined terminals
- Real-world example based on FEM
- Compilation can be a problem (RAM and time)
- Nice abstraction for defining the EDSL
- Excellent run time performance
- Writing expressions is easy
- Debugging can be tricky