Advanced Functional Programming with C++

Templates

Roy H. Stogner

The University of Texas at Austin

Feb 29, 2012
Outline

1. Stuff We Covered Last Time
   - Data Types
     - Multi-precision Verification
     - Array Operations
     - Automatic Differentiation
     - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Data Type Independence: Generic Programming

Example

template <int size, typename T>
class NumberArray {
    ........
    T dot(const NumericVector<T>& v) const {
        T r = 0;
        for(int i = 0; i != size; ++i)
            r += data[i] * v[i];
        return r;
    ........
    T* data;
};

NumericVector<float>
    floatvec;
NumericVector<complex<double> >
    complexdoublevec;
NumericVector<NumericVector<float> >
    floatmatrix;

- Template arguments can be integral or data types
- Class data members can depend on template argument
- Class methods can depend on template argument
- Templated classes can be instantiated with plain data types
  - ... or with other instantiated templated types
  - ... or with other instantiations of themselves!
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
     - Automatic Differentiation
     - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
     - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
“Dual Numbers”

For each simple operation \( s(x, y) \),

\[
s(a + b\epsilon, c + d\epsilon) = s(a, c) + \left( b \frac{\partial s}{\partial x}(a, c) + d \frac{\partial s}{\partial y}(a, c) \right) \epsilon
\]

Extending to division, algebraic functions, transcendental functions, etc also gives: for any function \( f(x) \),

\[
f(a + b\epsilon) = f(a) + bf'(a)\epsilon
\]

Composition of functions preserves these properties, so with independent variable \( x \in \mathbb{R} \), we set \( X \equiv x + \epsilon \) and get:

\[
f(X) = f(x) + f'(x)\epsilon
\]

Recursion gives us gradients, “hyper-duals”, etc.
Operator-Overloaded Forward Differentiation

Example

typedef double RawType
// typedef ShadowNumber<double, long double> RawType;
typedef DualNumber<RawType,RawType> FirstDerivType;

const FirstDerivType x(pi/6,1);  // Initializing independent var
const FirstDerivType sinx = sin(x);  // Caching just like normal
const FirstDerivType y = sinx*sinx;  // Smart pow would work too
const double raw_y = raw_value(y);  // No implicit down-conversions!
double deriv = raw_value(y.derivatives());
assert(deriv == 2*sin(pi/6)*cos(pi/6));  // FP ops match hand-code!
MASA PDE Examples

Manufactured Solution

```cpp
// Arbitrary manufactured solution
U.template get<0>() = u_0 + u_x * sin(a_u_x * PI * x / L) +
    u_y * cos(a_u_y * PI * y / L);
// Why not U[0] and U[1]? To be explained in this talk
U.template get<1>() = v_0 + v_x * cos(a_v_x * PI * x / L) +
    v_y * sin(a_v_y * PI * y / L);
ADScalar RHO = rho_0 + rho_x * sin(a_r_x * PI * x / L) +
    rho_y * cos(a_r_y * PI * y / L);
ADScalar P = p_0 + p_x * cos(a_p_x * PI * x / L) +
    p_y * sin(a_p_y * PI * y / L);
```

// Constitutive laws
Tensor GradU = gradient(U);
Tensor Tau = mu * (GradU + transpose(GradU) -
    2./3. * divergence(U) * RawArray::identity());
FullArray q = -k * T.derivatives();
MASA PDE Examples

Euler

// Gas state
ADScalar T = P / RHO / R;
ADScalar E = 1. / (Gamma-1.) * P / RHO;
ADScalar ET = E + .5 * U.dot(U);

// Conservation equation residuals
Scalar Q_rho = raw_value(divergence(RHO*U));
RawArray Q_rho_u = raw_value(divergence(RHO*U.outerproduct(U)) + P.derivatives());
Scalar Q_rho_e = raw_value(divergence((RHO*ET+P)*U));
MASA PDE Examples

Navier-Stokes

// Gas state
ADScalar T = P / RHO / R;
ADScalar E = 1. / (Gamma-1.) * P / RHO;
ADScalar ET = E + .5 * U.dot(U);

// Constitutive laws
Tensor GradU = gradient(U);
Tensor Tau = mu * (GradU + transpose(GradU) -
  2./3. * divergence(U) * RawArray:::identity());
FullArray q = -k * T.derivatives();

// Conservation equation residuals
Scalar Q_rho = raw_value(divergence(RHO*U));
RawArray Q_rho_u = raw_value(divergence(RHO*U.outerproduct(U) - Tau) +
  P.derivatives());
Scalar Q_rho_e = raw_value(divergence((RHO*ET+P)*U + q - Tau.dot(U)));

// Gas state
ADScalar T = P / RHO / R;
ADScalar E = 1. / (Gamma-1.) * P / RHO;
ADScalar ET = E + .5 * U.dot(U);

// Constitutive laws
Tensor GradU = gradient(U);
Tensor Tau = mu * (GradU + transpose(GradU) -
  2./3. * divergence(U) * RawArray:::identity());
FullArray q = -k * T.derivatives();

// Conservation equation residuals
Scalar Q_rho = raw_value(divergence(RHO*U));
RawArray Q_rho_u = raw_value(divergence(RHO*U.outerproduct(U) - Tau) +
  P.derivatives());
Scalar Q_rho_e = raw_value(divergence((RHO*ET+P)*U + q - Tau.dot(U)));
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Templates as “Functions”

- Classes take constant integers as template arguments
- Classes can “return” static const integers as members

Example

template<unsigned int N>
struct Factorial {
    static const unsigned int value = // Each value
        N * Factorial<N-1>::value; // is constant
};
template<>
struct Factorial<0> {
    static const unsigned int value = 1;
};

// Zero FP ops, zero tests/branches at runtime
unsigned int twelve = Factorial<4>::value;
Templates as “Functions”

- Classes take constant integers or types as template arguments
- Classes can “return” static const integers or typedef’ed types as members

Example

```cpp
template<typename T, typename S>
struct CompareTypes {
};

template <unsigned int size, typename T, typename T2, typename S2>
struct CompareTypes<NumberArray<size,T>,ShadowNumber<T2,S2> > {
    typedef NumberArray<size,
        typename CompareTypes<T,ShadowNumber<T2,S2> >::supertype>
    supertype;
};
```
Outline

1 Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2 Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3 Stuff We’ll Cover Next Time
Type Function Examples: “IfElse”

```cpp
template <bool Condition, typename TrueResult, typename FalseResult>
struct IfElse {
    typedef TrueResult type;
};

template <typename TrueResult, typename FalseResult>
struct IfElse<false, TrueResult, FalseResult> {
    typedef FalseResult type;
};

const bool imprecise = true; // compile-time constant
IfElse<imprecise,float,long double>::type one_third = 1.L/3.L;
```
Type Function Examples: “IfElse”

Example

```cpp
template <bool Condition, typename TrueResult, typename FalseResult>
struct IfElse {
    typedef TrueResult type;
};

template <typename TrueResult, typename FalseResult>
struct IfElse<false, TrueResult, FalseResult> {
    typedef FalseResult type;
};

const bool imprecise = true; // compile-time constant
IfElse<imprecise, float, long double>::type one_third = 1.L/3.L;
```
Type Function Examples: “TypesEqual”

Example

```cpp
template <typename T1, typename T2>
struct TypesEqual {
    static const bool value = false
};

template <typename T>
struct TypesEqual<T,T> {
    static const bool value = true
};
```
Type Function Examples: “enable_if”

- We partially specialized `CompareTypes` to handle combinations of classes and built-in types

```cpp
template <unsigned int size, typename T, typename T2>
struct CompareTypes<NumberArray<size,T>,T2>
{
    typedef NumberArray<size,typename CompareTypes<T,T2>::supertype> supertype;
};
```
Type Function Examples: “enable_if”

- We partially specialized `CompareTypes` to handle combinations of classes and built-in types.
- But we need to *limit* those specializations to built-in types.

**Example**

```cpp
template <unsigned int size, typename T, typename T2>
struct CompareTypes<NumberArray<size,T>, T2, typename boost::enable_if_c<IsBuiltin<T2>::value>::type>
{
    typedef NumberArray<size,typename CompareTypes<T,T2>::supertype>
    supertype;
};
```

Roy H. Stogner
Example

```cpp
namespace boost {

    template <bool B, class T = void>
    struct enable_if_c {
        typedef T type;
    };

    template <class T>
    struct enable_if_c<false, T> {};

    enable_if_c<true, double>::type one_third = 1.L/3.L;

```

Roy H. Stogner
namespace boost {
    template <bool B, class T = void>
    struct enable_if_c {
        typedef T type;
    };

    template <class T>
    struct enable_if_c<false, T> {};

    enable_if_c<true,double>::type one_third = 1.L/3.L;
    enable_if_c<false,double>::type compile_error = 0;
}
namespace boost {
    template <bool B, class T = void>
    struct enable_if_c {
        typedef T type;
    };

    template <class T>
    struct enable_if_c<false, T> {};

    enable_if_c<true,double>::type one_third = 1.L/3.L;
    enable_if_c<false,double>::type compile_error = 0;

    template <typename T>
    enable_if_c<IsBuiltin<T>::value,T> SFINAE(T& x) { return x*x; }
}
Outline

1 Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2 Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3 Stuff We’ll Cover Next Time
Data Structures with Templates: Type Container

Example

```cpp
template <typename HeadType, 
    typename TailContainer=NullContainer, 
    typename Comparison=ValueLessThan>
struct Container
{
    typedef HeadType head_type;
    typedef TailContainer tail_set;
    typedef Comparison comparison;
    // These may be empty types or may have data
    HeadType head;
    TailContainer tail;
};
```

```cpp
struct NullContainer
{
};
```
Data Structures with Templates: Type Container

Example

Container<IntType<1> > no_data_one_static_int;

Container<IntType<4>, NumberArray<3,double>> float_data_two_static_ints;

Container<IntType<1,double>, NumberArray<4,double>, NumberArray<3, int>> heterogenous_data;
Data Structures with Templates: Type Container

Example

Container<IntType<1> > no_data_one_static_int;
Container<IntType<1>,
    Container<IntType<4> > no_data_two_static_ints;
Data Structures with Templates: Type Container

Example

```plaintext
container<IntType<1>  >  no_data_one_static_int;
container<IntType<1>,
  container<IntType<4>  >  no_data_two_static_ints;
container<IntType<1,float>,
  container<IntType<4,float>  > float_data_two_static_ints;
```
Data Structures with Templates: Type Container

Example

```cpp
Container<IntType<1> > no_data_one_static_int;
Container<IntType<1>,
    Container<IntType<4> > no_data_two_static_ints;
Container<IntType<1,float>,
    Container<IntType<4,float> > float_data_two_static_ints;
Container<IntType<1,double>,
    Container<IntType<4>,NumberArray<3,double> > heterogenous_data;
```
Set Operations: Insert

Inserting into a `NullContainer` is easy:

- If we’re inserting null, we’re still left with a `NullContainer`
Set Operations: Insert

Inserting into a **NullContainer** is easy:

- If we’re inserting null, we’re still left with a **NullContainer**
- If not, we have a one-element container with the inserted element.
Set Operations: Insert

Inserting into a **NullContainer** is easy:

- If we’re inserting null, we’re still left with a **NullContainer**
- If not, we have a one-element container with the inserted element.
Set Operations: Insert

Inserting into a **NullContainer** is easy:

- If we’re inserting null, we’re still left with a **NullContainer**
- If not, we have a one-element container with the inserted element.

Example

```cpp
struct NullContainer
{
    ....
    template <typename ValueType, typename NewComparison>
    struct Insert
    {
        typedef
            typename IfElse<
                (TypesEqual<ValueType, NullContainer>::value),
                NullContainer,
                Container<ValueType, NullContainer, NewComparison>
            >::type type;
    
    };
    ....
};
```
Set Operations: Insert

Inserting in nonempty sorted sets is trickier:

• If we're inserting null, we return our same set.
• If we're inserting something smaller than our set head, then it becomes the new head and our set becomes the new tail.
• If we're inserting something larger than our set head, then the head stays unchanged and we insert into the tail.
• If we're inserting something equivalent to our set head, then the head may get "upgraded."
Set Operations: Insert

Inserting in nonempty sorted sets is trickier:

- If we’re inserting null, we return our same set
Set Operations: Insert

Inserting in nonempty sorted sets is trickier:

- If we’re inserting null, we return our same set
- If we’re inserting something smaller than our set head, then it becomes the new head and our set becomes the new tail
Set Operations: Insert

Inserting in nonempty sorted sets is trickier:

- If we’re inserting null, we return our same set
- If we’re inserting something smaller than our set head, then it becomes the new head and our set becomes the new tail
- If we’re inserting something larger than our set head, then the head stays unchanged and we insert into the tail
Set Operations: Insert

Inserting in nonempty sorted sets is trickier:

- If we’re inserting null, we return our same set
- If we’re inserting something smaller than our set head, then it becomes the new head and our set becomes the new tail
- If we’re inserting something larger than our set head, then the head stays unchanged and we insert into the tail
- If we’re inserting something equivalent to our set head, then the head may get “upgraded”
Set Operations: Insert

Example

```cpp
struct Container {
    template <typename ValueType, typename NewComparison>
    struct Insert {
        typedef
            typename IfElse<
            (TypesEqual<ValueType, NullContainer>::value),
            Sorted,
            typename IfElse<
            (Comparison::template LessThan<ValueType, typename Sorted::head_type>::value),
            Container<ValueType, Sorted
            >,
            typename IfElse<
            (Comparison::template LessThan<typename Sorted::head_type, ValueType>::value),
            Container<typename Sorted::head_type,
            typename Sorted::tail_set::template Insert<ValueType, NewComparison>::type
            >,
            Container<
            typename CombinedType<ValueType, typename Sorted::head_type>::type,
            typename Sorted::tail_set
            >
            >::type
            >::type
        >::type type;
    };
};
```
Set Operations: Union

- Union with an empty container is trivial.
Set Operations: Union

- Union with an empty container is trivial.
- With a non-empty container, `Insert` makes things easy. We just insert “one element at a time” recursively:

```cpp
template<typename Set2>
struct Union {
    typedef typename tail_set::Sorted::template Union<
        typename Set2::Sorted::template Insert<
            head_type
        >::type
    >::type type;
};
```
Set Operations: Union

- Union with an empty container is trivial.
- With a non-empty container, `Insert` makes things easy. We just insert “one element at a time” recursively:

```cpp
Example
template <typename Set2>
struct Union {
type;
};
```
Set Operations: Union

- Union with an empty container is trivial.
- With a non-empty container, Insert makes things easy. We just insert “one element at a time” recursively:

```cpp
template <typename Set2>
struct Union {
    typedef typename
        tail_set::Sorted::template Union<
            typename Set2::Sorted::template Insert<
                head_type
            >::type
        >::type type;
};
```
Set Operations: Intersection

- Intersection with an empty container is trivial.
Set Operations: Intersection

- Intersection with an empty container is trivial.
- With a non-empty container, `Contains` makes things easy. We selectively build a new set “one element at a time” recursively:
Set Operations: Intersection

- Intersection with an empty container is trivial.
- With a non-empty container, `Contains` makes things easy. We selectively build a new set “one element at a time” recursively:
Set Operations: Intersection

- Intersection with an empty container is trivial.
- With a non-empty container, `Contains` makes things easy. We selectively build a new set “one element at a time” recursively:

```cpp
template <typename Set2>
struct Intersection {
    typedef typename IfElse<Set2::template Contains<head_type>::value,
        Container<
            typename SymmetricCompareTypes<
                head_type,typename Set2::template Contains<head_type>::type
            >::supertype,
            typename tail_set::template Intersection<Set2>::type
        >,
        typename tail_set::template Intersection<Set2>::type
    >::type type;
};
```

Example
Set Operations: Difference

- Subtracting from an empty container is trivial.

```
template <typename Set2>
struct Difference
{
    typedef typename
    IfElse<Set2::template Contains<head_type>::value,
    typename tail_set::template Difference<Set2>::type,
    Container<head_type,
    typename tail_set::template Difference<Set2>::type,
    Comparison
    >::type
    >::type
    type;
};
```
Set Operations: Difference

- Subtracting from an empty container is trivial.
- With a non-empty container, `Contains` makes things easy. We selectively build a new set “one element at a time”, recursively:
Set Operations: Difference

- Subtracting from an empty container is trivial.
- With a non-empty container, \texttt{Contains} makes things easy. We selectively build a new set “one element at a time”, recursively:
Set Operations: Difference

- Subtracting from an empty container is trivial.
- With a non-empty container, Contains makes things easy. We selectively build a new set “one element at a time”, recursively:

Example

```cpp
template <typename Set2>
struct Difference
{
    typedef typename
        IfElse<Set2::template Contains<head_type>::value,
        typename tail_set::template Difference<Set2>::type,
        Container<head_type,
            typename tail_set::template Difference<Set2>::type,
            Comparison
        >
    >::type type;
};
```
Runtime Operations: ForEach

- Applying a standard functor to every set element
- Recursive compile-time metaprogram generates a simple linear run-time program
- \( f \) receives a non-compile-time-const \( v \)

Example

```cpp
struct RuntimeForEach
{
    template <typename Functor>
    void operator()(const Functor &f) {
        // \( f \) might want a reference, so we pass in a non-static copy
        const typename head_type::value_type v = head_type::value;
        f(v);
        typename tail_set::RuntimeForEach()(f);
    }
};
```
Runtime Operations: ForEach

- Applying a “template functor” to every set element
- Recursive compile-time metaprogram generates a simple linear run-time program
- \( f \) receives a compile-time-const \( \text{HeadType} \)

**Example**

```cpp
struct ForEach {
    template <typename Functor>
    void operator()(const Functor &f) {
        f.operator()<HeadType>(v);
        typename tail_set::ForEach()(f);
    }
};
```
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - **Sparse Data Structures**
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Operator-Overloaded Sparse Vectors

- An **IndexSet** lists sparse indices with potentially non-zero data
- e.g. the vector \([0 \ 0 \ 1 \ 0 \ 2 \ 0]\) will have \(\{3, 5\}\) or a superset as index set

Example

```cpp
template <typename T, typename IndexSet>
class SparseNumberArray {
public:
  typedef IndexSet index_set;

  static const unsigned int size = IndexSet::size;

  T& raw_at(unsigned int i) {
    return _data[i];
  }

private:
  T _data[size];
};
```
Operator-Overloaded Sparse Vectors

- An `IndexSet` lists sparse indices with potentially non-zero data
- Variable-index component access via runtime search through indices

Example

```cpp
template <typename T, typename IndexSet>
class SparseNumberArray {
public:
    typedef IndexSet index_set;

    static const unsigned int size = IndexSet::size;

    T& operator[](index_value_type i) {
        return _data[IndexSet::runtime_index_of(i)];
    }
};
```
Operator-Overloaded Sparse Vectors

- An IndexSet lists sparse indices with potentially non-zero data
- Variable-index component access via runtime search through indices
- Const-index $O(1)$ access via template arguments

Example

```cpp
template <typename T, typename IndexSet>
class SparseNumberArray {
public:
    typedef IndexSet index_set;

    static const unsigned int size = IndexSet::size;

    template <unsigned int i>
    typename entry_type<i>::type& get() {
        return _data[IndexSet::template IndexOf<i>::index];
    }

private:
    T _data[size];
};
```
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum
- Element-wise addition takes a Union of index sets
- Element-wise multiplication takes an Intersection of index sets
- Element-wise division takes the numerator’s index set, but asserts (compile-time!) that the divisor’s is a superset.
- Consider a typical binary function, max(a, b), applied element-wise to two sparse vectors:
  - For each element in the intersection of the index sets, we apply the function to the corresponding pair of vector data.
  - For each element in each asymmetric difference of the index sets, we apply the function to a pair of the present vector datum and 0.
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum

Other operations require simple set operations:

- Element-wise addition takes a Union of index sets
- Element-wise multiplication takes an Intersection of index sets
- Element-wise division takes the numerator’s index set, but asserts (compile-time!) that the divisor’s is a superset.

- Consider a typical binary function, \(\text{max}(a, b)\), applied element-wise to two sparse vectors:
  - For each element in the intersection of the index sets, we apply the function to the corresponding pair of vector data.
  - For each element in each asymmetric difference of the index sets, we apply the function to a pair of the present vector datum and 0.
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum

Other operations require simple set operations:

- Element-wise addition takes a *Union* of index sets
- Element-wise multiplication takes an *Intersection* of index sets
- Element-wise division takes the numerator’s index set, but asserts (compile-time!) that the divisor’s is a superset.
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum

Other operations require simple set operations:

- Element-wise addition takes a Union of index sets
- Element-wise multiplication takes an Intersection of index sets
- Element-wise division takes the numerator’s index set, but asserts (compile-time!) that the divisor’s is a superset.

Others require repeated set operations:
Operator-Overloaded Sparse Vector Operations

Some operations are easily defined, no index set operations needed:

- Multiplication by a constant: for loop over all data, multiply datum

Other operations require simple set operations:

- Element-wise addition takes a \texttt{Union} of index sets
- Element-wise multiplication takes an \texttt{Intersection} of index sets
- Element-wise division takes the numerator’s index set, but asserts (compile-time!) that the divisor’s is a superset.

Others require repeated set operations:

- Consider a typical binary function, \texttt{max(a,b)}, applied element-wise to two sparse vectors:
- For each element in the intersection of the index sets, we apply the function to the corresponding pair of vector data.
- For each element in each asymmetric difference of the index sets, we apply the function to a pair of the present vector datum and 0.
Operator-Overloaded Sparse Vector Functions

Example

```cpp
template <typename T, typename T2, typename IndexSet, typename IndexSet2>
inline SparseNumberArray<typename SymmetricCompareTypes<T,T2>::supertype,
                         typename IndexSet::template Union<IndexSet2>::type>
funcname (const SparseNumberArray<T, IndexSet>& a, const SparseNumberArray<T2, IndexSet2>& b)
{
    typedef typename SymmetricCompareTypes<T,T2>::supertype TS;
    typedef typename IndexSet::template Union<IndexSet2>::type IndexSetS;
    SparseNumberArray<TS, IndexSetS> returnval;

    typename IndexSet::template Intersection<IndexSet2>::type::ForEach()
    (BinaryArrayFunctor<std::binary_function<TS,TS,TS>,IndexSet,IndexSet2,IndexSet,T,T2,TS>
    (a.raw_data(), b.raw_data(), returnval.raw_data(), std::ptr_fun(std::funcname<TS>)));
    typename IndexSet::template Difference<IndexSet2>::type::ForEach()
    (UnaryArrayFunctor<std::unary_function<TS,TS>,IndexSet,T,TS>
    (a.raw_data(), returnval.raw_data(), std::bind2nd(std::ptr_fun(std::funcname<TS>),0)));
    typename IndexSet2::template Difference<IndexSet>::type::ForEach()
    (UnaryArrayFunctor<std::unary_function<TS,TS>,IndexSet2,T,TS>
    (b.raw_data(), returnval.raw_data(), std::bind1st(std::ptr_fun(std::funcname<TS>),0)));

    return returnval;
}
```
Operator-Overloaded Sparse Structures
Consider recursion with dual numbers, for automatic differentiation
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- `DualNumber x` and `y` are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

\[
\begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 & 0 \end{bmatrix}
\]

\[
\begin{bmatrix} \cdot \cdot \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} + \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}
\]

- From 6 FLOPs down to 0!
- But that was just practice!
- What we want is sparse structures-of-everything!
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- \texttt{DualNumber} $x$ and $y$ are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- DualNumber $x$ and $y$ are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

$$\begin{bmatrix} 0 & 0 & 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 2 & 0 \end{bmatrix}$$
Operator-Overloaded Sparse Structures
Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- `DualNumber x` and `y` are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

\[
\begin{bmatrix}
0 & 0 & 0 & 2 & 0 \\
0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
+ 
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 2 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & 1 & 0 & 2 \\
0 & 0 & 1 & 0 & 2 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdots & \cdots & 2 & \cdots \\
\cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}
+ 
\begin{bmatrix}
\cdots & 1 & \cdots \\
\cdots & \cdots & \cdots \\
\end{bmatrix}
\]
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- \( x \) is only non-constant w.r.t. \( x \); \( y \) is only non-constant w.r.t. \( y \)
- \texttt{DualNumber} \( x \) and \( y \) are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 2 & 0 \\
\end{bmatrix}
+ \begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 \\
\end{bmatrix}
= \begin{bmatrix}
0 & 0 & 1 & 0 & 2 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdots & \cdots & 2 & \cdots \\
\end{bmatrix}
+ \begin{bmatrix}
\cdots & 1 & \cdots \\
\end{bmatrix}
= \begin{bmatrix}
\cdots & 1 & 2 & \cdots \\
\end{bmatrix}
\]
Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- `DualNumber x` and `y` are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \cdots & \cdots & 2 \cdot \end{bmatrix} + \begin{bmatrix} \cdots & 1 & \cdots \end{bmatrix} = \begin{bmatrix} \cdots & 1 & 2 \cdot \end{bmatrix}$$

- From 6 FLOPs down to 0!
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- \( x \) is only non-constant w.r.t. \( x \); \( y \) is only non-constant w.r.t. \( y \)
- `DualNumber x` and `y` are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 2 & 0
\end{bmatrix} + \begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix} = \begin{bmatrix}
0 & 0 & 1 & 0 & 2 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdots & \cdots & 2 & \cdots
\end{bmatrix} + \begin{bmatrix}
\cdots & 1 & \cdots
\end{bmatrix} = \begin{bmatrix}
\cdots & 1 & 2 & \cdots
\end{bmatrix}
\]

- From 6 FLOPs down to 0!
- But that was just *practice*!
Operator-Overloaded Sparse Structures

Consider recursion with dual numbers, for automatic differentiation

- $x$ is only non-constant w.r.t. $x$; $y$ is only non-constant w.r.t. $y$
- `DualNumber x` and `y` are given unit vector gradients
- Their sums, products, etc. then all have optimally sparse gradients.

We now have excellent sparse vectors-of-T!

\[
\begin{bmatrix} 0 & 0 & 0 & 2 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 & 0 \end{bmatrix}
\]

\[
\begin{bmatrix} \cdots & \cdots & 2 & \cdots \end{bmatrix} + \begin{bmatrix} \cdots & 1 & \cdots \end{bmatrix} = \begin{bmatrix} \cdots & 1 & 2 & \cdots \end{bmatrix}
\]

- From 6 FLOPs down to 0!
- But that was just *practice*!
- What we want is sparse structures-of-everything!
Operator-Overloaded Sparse Structures
Consider recursion with arrays, for matrices, hessians
Operator-Overloaded Sparse Structures

Consider recursion with arrays, for matrices, hessians

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
+ \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0
\end{bmatrix}
= \begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
Operator-Overloaded Sparse Structures
Consider recursion with arrays, for matrices, hessians

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix} + \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
\end{bmatrix} = \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 2 & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\end{bmatrix} + \begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot \\
\end{bmatrix} = \]

A SparseNumberArray -of- SparseNumberArray is sparse...
But only with a tensor-product sparsity structure.

▶ Triangular matrices become full?
▶ Diagonal matrices become full??
Operator-Overloaded Sparse Structures

Consider recursion with arrays, for matrices, hessians

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
+ 
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & 2 & \cdot \\
\cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{bmatrix}
+ 
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
1 & \cdot & \cdot \\
\cdot & \cdot & \cdot \\
\end{bmatrix}
= 
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
0 & 2 & \cdot \\
\cdot & \cdot & \cdot \\
1 & 0 & \cdot \\
\end{bmatrix}
\]
Operator-Overloaded Sparse Structures

Consider recursion with arrays, for matrices, hessians

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
+ 
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & 2 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}
+ 
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}
= 
\begin{bmatrix}
\cdot & \cdot & \cdot & \cdot \\
0 & 2 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot \\
1 & 0 & \cdot & \cdot \\
\end{bmatrix}
\]

- A SparseNumberArray-of-SparseNumberArray is sparse...
- But only with a tensor-product sparsity structure.
  - Triangular matrices become full?
  - Diagonal matrices become full??
Operator-Overloaded Sparse Structures

- An `IndexSet` stores data, not just static indices

Example

```cpp
template <typename IndexSet>
class SparseNumberStruct {
public:
    typedef IndexSet index_set;

    static const unsigned int size = IndexSet::size;

    template <unsigned int i>
        typename entry_type<i>::type& get() {
            return _data.template data<UnsignedIntType<i> >(); }

private:
    IndexSet _data;
};
```
Operator-Overloaded Sparse Structures

- An `IndexSet` stores data, not just static indices
- Index-dependent data type makes variable-index component access impossible!

**Example**

```cpp
template <typename IndexSet>
class SparseNumberStruct {
public:
    typedef IndexSet index_set;

    static const unsigned int size = IndexSet::size;

    template <unsigned int i>
    typename entry_type<i>::type& get() {
        return _data.template data<UnsignedIntType<i> >();
    }

private:
    IndexSet _data;
};
```
Operator-Overloaded Sparse Structures

- Now we can do true sparse matrices!

Example

```cpp
template <typename Set1>
struct SetDiagonalTensor
{
    typedef
        Container<
            typename Set1::head_type::template rebind<
                SparseNumberStruct<
                    Container<typename Set1::head_type, NullContainer>
                >
            >::other,
            typename SetDiagonalTensor<typename Set1::tail_set>::type
        >
    >::other,
    typename SetDiagonalTensor<
        typename Set1::head_type
    >::type
    > type;
};
```
Operator-Overloaded Sparse Structures

- Now we can do true sparse matrices!
- Example: a vector of different unit vectors gives a diagonal tensor type

Example

```cpp
template <typename Set1>
struct SetDiagonalTensor
{
  typedef
    typename Set1::head_type::template rebind<
    SparseNumberStruct<
        Container<
            typename Set1::head_type::template rebind<
                SparseNumberStruct<
                    Container<typename Set1::head_type, NullContainer>
                >
            >
        >::other,
        typename SetDiagonalTensor<typename Set1::tail_set>::type
      >::other,
    typename SetDiagonalTensor<typename Set1::tail_set>::type
  >
      type;
};
```
Operator-Overloaded Sparse Structure Operations

No operations are easily defined.
Operator-Overloaded Sparse Structure Operations

No operations are easily defined.

- C for loops don’t work over heterogenous structures.
Operator-Overloaded Sparse Structure Operations

No operations are easily defined.

- C `for` loops don’t work over heterogenous structures.
- Templated functors must be defined for every operation.

```cpp
template <typename T2, typename IndexSet>
inline typename MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype
operator * (const SparseNumberStruct<IndexSet>& a, const T2& b)
{
    typedef typename MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype type;
    typename CompareTypes<SparseNumberStruct<IndexSet>,T2>::supertype returnval;
    typename IndexSet::ForEach()
        (BinaryFunctor<MultipliesSubfunctor, IndexSet, ConstantDataSet<T2>, typename type::index_set>
            (MultipliesSubfunctor(), a.raw_data(), ConstantDataSet<T2>(b), returnval.raw_data()));
    return returnval;
}
```
Operator-Overloaded Sparse Structure Operations

No operations are easily defined.

- C `for` loops don’t work over heterogenous structures.
- Templated functors must be defined for every operation.

```
Example:
```
Operator-Overloaded Sparse Structure Operations

*No* operations are easily defined.

- C `for` loops don’t work over heterogenous structures.
- Templated functors must be defined for every operation.

Multiplication by a constant, using a functor and subfunctor:

```cpp
Example
template <typename T2, typename IndexSet>
inline
type = MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype
operator * (const SparseNumberStruct<IndexSet>& a, const T2& b)
{
    typedef typename MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype type;
    typename CompareTypes<SparseNumberStruct<IndexSet>,T2>::supertype
    returnval;
    typename IndexSet::ForEach()
    (BinaryFunctor<MultipliesSubfunctor, IndexSet, ConstantDataSet<T2>, typename type::index_set>
    (MultipliesSubfunctor(), a.raw_data(), ConstantDataSet<T2>(b), returnval.raw_data()));
    return returnval;
}
```
Operator-Overloaded Sparse Structure Operations

No operations are easily defined.

- C for loops don’t work over heterogenous structures.
- Templated functors must be defined for every operation.

Multiplication by a constant, using a functor and subfunctor:

Example

```cpp
template <typename T2, typename IndexSet>
inline
typename MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype
operator * (const SparseNumberStruct<IndexSet>& a, const T2& b)
{
    typedef typename MultipliesType<SparseNumberStruct<IndexSet>,T2>::supertype type;

    typename CompareTypes<SparseNumberStruct<IndexSet>,T2>::supertype
        returnval;
    typename IndexSet::ForEach()
        (BinaryFunctor<MultipliesSubfunctor, IndexSet, ConstantDataSet<T2>, typename type::index_set>
            (MultipliesSubfunctor(), a.raw_data(), ConstantDataSet<T2>(b), returnval.raw_data()));
    return returnval;
}
```
Templated Functors

- A functor class datatype represents an assignment equation
Templated Functors

- A functor class datatype represents an assignment equation
- Template overloading of `operator()` accepts arbitrary input types

```cpp
struct MultipliesSubfunctor {
  template <typename T1, typename T2>
  struct Return {
    typedef typename SymmetricMultipliesType<T1,T2>::supertype type;
  }

  template <typename T1, typename T2>
  typename Return<T1,T2>::type operator()(const T1& x, const T2& y) const {
    return x * y;
  }
};
```
Templated Functors

- A functor class datatype represents an assignment equation
- Template overloading of `operator()` accepts arbitrary input types
- `CompareTypes` or similar classes (or C++11 `decltype`) provides proper return types

```cpp
struct MultipliesSubfunctor {
    template <typename T1, typename T2>
    struct Return {
        typedef typename SymmetricMultipliesType<T1,T2>::supertype type;
    };
    template <typename T1, typename T2>
    typename Return<T1,T2>::type operator()(const T1& x, const T2& y) const { return x * y; }
};
```
Templated Functors

- A functor class datatype represents an assignment equation
- Template overloading of `operator()` accepts arbitrary input types
- `CompareTypes` or similar classes (or C++11 `decltype`) provides proper return types
Templated Functors

- A functor class datatype represents an assignment equation
- Template overloading of `operator()` accepts arbitrary input types
- `CompareTypes` or similar classes (or C++11 `decltype`) provides proper return types

Example

```cpp
struct MultipliesSubfunctor {
    template <typename T1, typename T2>
    struct Return {
        typedef typename
            SymmetricMultipliesType<T1,T2>::supertype type;
    };

    template <typename T1, typename T2>
    typename Return<T1,T2>::type
    operator()(const T1& x, const T2& y) const { return x * y; }
};
```
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Equations

- A templated functor is almost a full physics equation!
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
    - Enumerated types are metaprogramming compatible
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
  - Enumerated types are metaprogramming compatible
  - Equations can be split into minimal chunks for modularity
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
    - Enumerated types are metaprogramming compatible
  - Equations can be split into minimal chunks for modularity
  - Equations *must* be split into minimal chunks for Jacobian optimization?
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
    - Enumerated types are metaprogramming compatible
  - Equations can be split into minimal chunks for modularity
  - Equations *must* be split into minimal chunks for Jacobian optimization?

```cpp
#define DeclareUnaryPhysics(physicsname, input1enum, outputenum, code) 
struct physicsname {
  static const char* name() { return #physicsname; }
  typedef UIntSetConstructor<input1enum>::type inputset;
  typedef UIntSetConstructor<outputenum>::type outputset;

  template <typename inputtype> 
  auto operator() (const inputtype& input1enum##_NAME) -> decltype(code) {
    return code;
  }
};
```
Equations

- A templated functor is almost a full physics equation!
  - Define *which* physical quantities are inputs and outputs
    - Enumerated types are metaprogramming compatible
  - Equations can be split into minimal chunks for modularity
  - Equations *must* be split into minimal chunks for Jacobian optimization?

Example

```cpp
#define DeclareUnaryPhysics(physicsname, input1enum, outputenum, code) \
struct physicsname { \
    static const char* name() { return #physicsname; } \
    typedef UIntSetConstructor<input1enum>::type inputset; \
    typedef UIntSetConstructor<outputenum>::type outputset; \
\} \

    template <typename inputtype> \
    auto operator()(const inputtype& input1enum##_NAME) \
        -> decltype(code) { \
        return code; \
    } \
}
```
Equations

Example

DeclareUnaryPhysics(DensityFromSpeciesDensities, DENSITIES_VAR, DENSITY_VAR, rhoi.sum());

DeclareBinaryPhysics(MomentumFromVelocity, DENSITY_VAR, VELOCITY_VAR, MOMENTUM_VAR, U * rho);

DeclareBinaryPhysics(VelocityFromMomentum, DENSITY_VAR, MOMENTUM_VAR, VELOCITY_VAR, rhoU / rho);

DeclareBinaryPhysics(SpecificEnergyFromSpecificInternalEnergy, SPECIFIC_INTERNAL_ENERGY_VAR, SPEED_SQUARED_VAR, SPECIFIC_ENERGY_VAR, e + UdotU / 2);
Equations

Example

```java
DeclareUnaryPhysics(DensityFromSpeciesDensities,
    DENSITIES_VAR, DENSITY_VAR,
    rhoi.sum());

DeclareBinaryPhysics(MomentumFromVelocity,
    DENSITY_VAR, VELOCITY_VAR, MOMENTUM_VAR,
    U * rho);

DeclareBinaryPhysics(VelocityFromMomentum,
    DENSITY_VAR, MOMENTUM_VAR, VELOCITY_VAR,
    rhoU / rho);

DeclareBinaryPhysics(SpecificEnergyFromSpecificInternalEnergy,
    SPECIFIC_INTERNAL_ENERGY_VAR, SPEED_SQUARED_VAR,
    SPECIFIC_ENERGY_VAR,
    e + UdotU / 2);
```

- Functors get reused in every Physics set, for all input argument types
Equations

Example

```java
DeclareUnaryPhysics(DensityFromSpeciesDensities,
    DENSITIES_VAR, DENSITY_VAR,
    rhoi.sum());
DeclareBinaryPhysics(MomentumFromVelocity,
    DENSITY_VAR, VELOCITY_VAR, MOMENTUM_VAR,
    U * rho);
DeclareBinaryPhysics(VelocityFromMomentum,
    DENSITY_VAR, MOMENTUM_VAR, VELOCITY_VAR,
    rhoU / rho);
DeclareBinaryPhysics(SpecificEnergyFromSpecificInternalEnergy,
    SPECIFIC_INTERNAL_ENERGY_VAR, SPEED_SQUARED_VAR,
    SPECIFIC_ENERGY_VAR,
    e + UdotU / 2);
```

- Functors get reused in every Physics set, for all input argument types
- Minimizing how much code must be entrusted to people like Marco
Equation Sets

- Equations are now represented by data types

```cpp
typedef VectorConstructor<
  DensityFromSpeciesDensities,
  VelocityFromMomentum,
  MomentumFromVelocity,
  SpeedSquaredFromVelocity,
  SpecificEnergyFromConserved,
  SpecificInternalEnergyFromSpecificEnergy,
  SpecificEnergyFromSpecificInternalEnergy,
  LinearTranslationalRotationalEnergyFromTemperature,
  SpecificInternalEnergyFromOnlyTransationalRotationalEnergy
>::type AllPhysics;
```
Equation Sets

- Equations are now represented by data types
- Data types can be stuck into metaprogramming containers!
Equation Sets

- Equations are now represented by data types
- Data types can be stuck into metaprogramming containers!

Example:

```
typedef VectorConstructor<
  DensityFromSpeciesDensities,
  VelocityFromMomentum,
  MomentumFromVelocity,
  SpeedSquaredFromVelocity,
  SpecificEnergyFromConserved,
  SpecificInternalEnergyFromSpecificEnergy,
  SpecificEnergyFromSpecificInternalEnergy,
  LinearTranslationalRotationalEnergyFromTemperature,
  SpecificInternalEnergyFromOnlyTransationalRotationalEnergy
>::type AllPhysics;
```
Equation Sets

- Equations are now represented by data types
- Data types can be stuck into metaprogramming containers!

Example

```cpp
typedef VectorConstructor<
    DensityFromSpeciesDensities,
    VelocityFromMomentum,
    MomentumFromVelocity,
    SpeedSquaredFromVelocity,
    SpecificEnergyFromConserved,
    SpecificInternalEnergyFromSpecificEnergy,
    SpecificEnergyFromSpecificInternalEnergy,
    LinearTranslationalRotationalEnergyFromTemperature,
    SpecificInternalEnergyFromOnlyTransationalRotationalEnergy
>::type AllPhysics;
```
Variable Sets

- Variables are now represented by enumerated values
Variable Sets

- Variables are now represented by enumerated values
- Enums can be stuck into metaprogramming containers!

```cpp
typedef UIntSetConstructor<
    DENSITIES_VAR,
    VELOCITY_VAR,
    TEMPERATURE_VAR
>::type PrimitiveVars;

typedef UIntSetConstructor<
    DENSITIES_VAR,
    MOMENTUM_VAR,
    ENERGY_VAR
>::type ConservedVars;
```
Variable Sets

- Variables are now represented by enumerated values
- Enums can be stuck into metaprogramming containers!
**Variable Sets**

- Variables are now represented by enumerated values
- Enums can be stuck into metaprogramming containers!

### Example

```cpp
typedef UIntSetConstructor<
    DENSITIES_VAR,
    VELOCITY_VAR,
    TEMPERATURE_VAR
>::type PrimitiveVars;

typedef UIntSetConstructor<
    DENSITIES_VAR,
    MOMENTUM_VAR,
    ENERGY_VAR
>::type ConservedVars;
```
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
- Consider each equation as a black node
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
- Consider each equation as a black node
  - Each output is a directed edge from equation to variable
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
- Consider each equation as a black node
  - Each output is a directed edge from equation to variable
  - Each input is a directed edge from variable to equation
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
- Consider each equation as a black node
  - Each output is a directed edge from equation to variable
  - Each input is a directed edge from variable to equation
Equation Sets as Bicolored Graphs

- Consider each variable as a red node
- Consider each equation as a black node
  - Each output is a directed edge from equation to variable
  - Each input is a directed edge from variable to equation

Then our physics now looks like a graph!
Equation Sets as Bicolored Graphs

We can use metaprogramming to answer questions about the graph.

\[ \rho E \]

\[ \rho e \]

\[ (\rho e)_i \]

\[ e_i \]

\[ \rho \]

\[ \vec{u} \cdot \vec{u} \]

\[ \vec{u} \]

\[ \rho i \]

\[ T \]
Equation Sets as Bicolored Graphs

We can use metaprogramming to answer questions about the graph.

- Given a set of input variables:
  - Which non-input variables in the graph are now determined?
  - What are the dependencies of each variable?

- And a set of output variables:
  - Which equations must be solved to evaluate them?
  - In which order?
Outline

1. Stuff We Covered Last Time
   - Data Types
   - Multi-precision Verification
   - Array Operations
   - Automatic Differentiation
   - Functional Metaprogramming with Templates

2. Numeric Functional Programming
   - Advanced Functional Programming with Templates
   - Functional Data Structures
   - Sparse Data Structures
   - Physics as a Graph Problem
   - Evaluations on the Graph
   - Differentiation on the Graph

3. Stuff We’ll Cover Next Time
Dependency Checking

• Which variables depend on which other variables?
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices
- We can `DependencyInsert` a new target variable plus its dependency variables into a dependency set
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices

- We can `DependencyInsert` a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices

- We can \texttt{DependencyInsert} a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable

- Find any target in the dependency set which depends on the new variable
  - It now also depends on the new variable's (old and new) dependencies

- We can greedily add solvable equations to a dependency set, one at a time
  - Initial dependency set targets are users' inputs, with no dependencies
  - An equation is "solvable" when all its inputs are in the dependency set and its output is not
  - Greedy algorithm handles cyclic graphs!
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices

- We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable
  - Find any target in the dependency set which depends on the new variable
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices

- We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable
  - Find any target in the dependency set which depends on the new variable
    - It now also depends on the new variable’s (old and new) dependencies
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices
- We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable
  - Find any target in the dependency set which depends on the new variable
    - It now also depends on the new variable’s (old and new) dependencies
- We can greedily add solvable equations to a dependency set, one at a time
Dependency Checking

• Which variables depend on which other variables?
  ▶ A dependency set is expressable as an indexed set-of-sets
  ▶ Each target index corresponds to a set of dependency indices

• We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  ▶ Find any target in the dependency set which is also a new dependency
    • It and its current dependencies become new dependencies of the new variable
  ▶ Find any target in the dependency set which depends on the new variable
    • It now also depends on the new variable’s (old and new) dependencies

• We can greedily add solvable equations to a dependency set, one at a time
  ▶ Initial dependency set targets are users’ inputs, with no dependencies
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices
- We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable
  - Find any target in the dependency set which depends on the new variable
    - It now also depends on the new variable’s (old and new) dependencies
- We can greedily add solvable equations to a dependency set, one at a time
  - Initial dependency set targets are users’ inputs, with no dependencies
  - An equation is “solvable” when all its inputs are in the dependency set and its output is not
Dependency Checking

- Which variables depend on which other variables?
  - A dependency set is expressable as an indexed set-of-sets
  - Each target index corresponds to a set of dependency indices
- We can **DependencyInsert** a new target variable plus its dependency variables into a dependency set
  - Find any target in the dependency set which is also a new dependency
    - It and its current dependencies become new dependencies of the new variable
  - Find any target in the dependency set which depends on the new variable
    - It now also depends on the new variable’s (old and new) dependencies
- We can greedily add solvable equations to a dependency set, one at a time
  - Initial dependency set targets are users’ inputs, with no dependencies
  - An equation is “solvable” when all its inputs are in the dependency set and its output is not
  - Greedy algorithm handles cyclic graphs!
Dependency Checking

Example

template <typename prev_dependencies, typename new_dependency>
struct DependencyInsert
{
    typedef typename prev_dependencies::template
        Intersection<typename new_dependency::data_type>::type recursing_dependencies;

    typedef typename new_dependency::template rebind<
        typename new_dependency::data_type::template Union<
            typename SetOfSetsUnion<recursing_dependencies>::type
        >::type
    >::other complete_new_dependency;

    typedef Container<
        typename IfElse<
            typename prev_dependencies::head_type::data_type::template
            Contains<typename new_dependency>::value,
            typename prev_dependencies::head_type::template rebind<
                typename prev_dependencies::head_type::data_type::template
                Union<typename complete_new_dependency::data_type>::type
            >::other,
            typename prev_dependencies::head_type
        >::type,
        typename DependencyInsert<typename prev_dependencies::tail_set,
            complete_new_dependency>::type
    > type;
};
Given input variables, output variables, and equation functor class types connecting them, we want a **SolveList**: an ordered set of which equations to solve in what order.
Given input variables, output variables, and equation functor class types connecting them, we want a **SolveList**: an ordered set of which equations to solve in what order.

1. Find all the dependencies of your output variables
SolveList

Given input variables, output variables, and equation functor class types connecting them, we want a **SolveList**: an ordered set of which equations to solve in what order.

1. **Find all the dependencies of your output variables**
   - Unreachable output variables become a compile-time error

2. **Greedy search through equations (in the same order as for dependency finding!), adding solveable equations to list**

---

Roy H. Stogner

Generic

Feb 29, 2012 48 / 56
**SolveList**

Given input variables, output variables, and equation functor class types connecting them, we want a **SolveList**: an ordered set of which equations to solve in what order.

1. Find all the dependencies of your output variables
   - Unreachable output variables become a compile-time error
   - Unnecessary non-input non-output variables are left out
Given input variables, output variables, and equation functor class types connecting them, we want a **SolveList**: an ordered set of which equations to solve in what order.

1. Find all the dependencies of your output variables
   - Unreachable output variables become a compile-time error
   - Unnecessary non-input non-output variables are left out

2. Greedy search through equations (in the same order as for dependency finding!), adding solveable equations to list
State Updating

With user-specified types on all variables, \texttt{SolveList} evaluation is easy:
State Updating

With user-specified types on all variables, SolveList evaluation is easy:

1. Create a Container type holding the specified variable types
State Updating

With user-specified types on all variables, \texttt{SolveList} evaluation is easy:

1. Create a \texttt{Container} type holding the specified variable types
2. Instantiate this \texttt{State state}; set values on the user input variables
State Updating

With user-specified types on all variables, `SolveList` evaluation is easy:

1. Create a `Container` type holding the specified variable types
2. Instantiate this `State state`; set values on the user input variables
3. Do a `ForEach` loop over the `SolveList`
State Updating

With user-specified types on all variables, `SolveList` evaluation is easy:

1. Create a `Container` type holding the specified variable types
2. Instantiate this `State state`; set values on the user input variables
3. Do a `ForEach` loop over the `SolveList`
   - Pass input variables from `state`, assign output variables to `state` for each equation
State Updating

With user-specified types on all variables, `SolveList` evaluation is easy:

1. Create a `Container` type holding the specified variable types
2. Instantiate this `State state`; set values on the user input variables
3. Do a `ForEach` loop over the `SolveList`
   - Pass input variables from `state`, assign output variables to `state` for each equation
State Updating

With user-specified types on all variables, SolveList evaluation is easy:

1. Create a Container type holding the specified variable types
2. Instantiate this State state; set values on the user input variables
3. Do a ForEach loop over the SolveList
   - Pass input variables from state, assign output variables to state for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:
State Updating

With user-specified types on all variables, SolveList evaluation is easy:

1. Create a Container type holding the specified variable types
2. Instantiate this State state; set values on the user input variables
3. Do a ForEach loop over the SolveList
   - Pass input variables from state, assign output variables to state for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:

1. Create a Container type holding the specified input variable types
State Updating

With user-specified types on all variables, \texttt{SolveList} evaluation is easy:

1. Create a \texttt{Container} type holding the specified variable types
2. Instantiate this \texttt{State state}; set values on the user input variables
3. Do a \texttt{ForEach} loop over the \texttt{SolveList}
   ▶ Pass input variables from \texttt{state}, assign output variables to \texttt{state} for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:

1. Create a \texttt{Container} type holding the specified input variable types
2. Do a metaprogramming loop over the \texttt{SolveList}
State Updating

With user-specified types on all variables, $\text{SolveList}$ evaluation is easy:

1. Create a $\text{Container}$ type holding the specified variable types
2. Instantiate this $\text{State state}$; set values on the user input variables
3. Do a $\text{ForEach}$ loop over the $\text{SolveList}$
   - Pass input variables from $\text{state}$, assign output variables to $\text{state}$ for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:

1. Create a $\text{Container}$ type holding the specified input variable types
2. Do a metaprogramming loop over the $\text{SolveList}$
   - Pass input variable types to each equation
State Updating

With user-specified types on all variables, SolveList evaluation is easy:

1. Create a Container type holding the specified variable types
2. Instantiate this State state; set values on the user input variables
3. Do a ForEach loop over the SolveList
   - Pass input variables from state, assign output variables to state for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:

1. Create a Container type holding the specified input variable types
2. Do a metaprogramming loop over the SolveList
   - Pass input variable types to each equation
   - Insert output variable types into the State
State Updating

With user-specified types on all variables, \textit{SolveList} evaluation is easy:

1. Create a \texttt{Container} type holding the specified variable types
2. Instantiate this \texttt{State state}; set values on the user input variables
3. Do a \texttt{ForEach} loop over the \textit{SolveList}
   - Pass input variables from \texttt{state}, assign output variables to \texttt{state} for each equation

With user-specified types on input variables only, we first need a metaprogramming pass:

1. Create a \texttt{Container} type holding the specified input variable types
2. Do a metaprogramming loop over the \textit{SolveList}
   - Pass input variable types to each equation
   - Insert output variable types into the \texttt{State}
3. Do a \texttt{ForEach} loop at runtime as before
Outline

1 Stuff We Covered Last Time
   • Data Types
   • Multi-precision Verification
   • Array Operations
   • Automatic Differentiation
   • Functional Metaprogramming with Templates

2 Numeric Functional Programming
   • Advanced Functional Programming with Templates
   • Functional Data Structures
   • Sparse Data Structures
   • Physics as a Graph Problem
   • Evaluations on the Graph
   • Differentiation on the Graph

3 Stuff We’ll Cover Next Time
Equation Derivatives

Differentiation looks already solved!

1. Collect your Physics
Equation Derivatives

Differentiation looks already solved!

1. Collect your Physics
2. Choose your input variables
Equation Derivatives

Differentiation looks already solved!

1. Collect your Physics
2. Choose your input variables
3. Evaluate the SolveList
Equation Derivatives

Differentiation looks already solved!

1. Collect your Physics
2. Choose your input variables
3. Evaluate the SolveList
4. Use DualNumber input variable types
Equation Derivatives

Differentiation looks already solved!

1. Collect your **Physics**
2. Choose your input variables
3. Evaluate the **SolveList**
4. Use **DualNumber** input variable types
5. Take **derivatives()** of output variables
Equation Derivatives

Differentiation looks already solved!

1. Collect your Physics
2. Choose your input variables
3. Evaluate the SolveList
4. Use DualNumber input variable types
5. Take derivatives() of output variables
6. Brag
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs or subgraphs!
  2. Adjoint differentiation issues:
     1. Needlessly expensive for many-output-variable, few-input variable graphs or subgraphs!
     2. Typically requires additional temporary data, particularly with operator overloading...
     3. No nice *DualNumber* representation?
  3. Optimal Jacobian Accumulation issues:
     1. NP-complete!
        - don't want to stress-test compilers...
     2. Solution: “Meh, Good Enough” Jacobian Accumulation
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs

- Adjoint differentiation issues:
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  2. Typically requires additional temporary data
  3. No nice `DualNumber` representation?

- Optimal Jacobian Accumulation issues:
  1. NP-complete!

Solution: “Meh, Good Enough” Jacobian Accumulation
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  - Needlessly expensive for many-input-variable, few-output variable graphs
    - or subgraphs!

- **Adjoint differentiation issues:**
  - Needlessly expensive for many-output-variable, few-input variable graphs
  - Typically requires additional temporary data
    - particularly with operator overloading...
  - No nice [DualNumber](#) representation?

- **Optimal Jacobian Accumulation issues:**
  - NP-complete!
    - don't want to stress-test compilers...
  - Solution: “Meh, Good Enough” Jacobian Accumulation
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs
     - or subgraphs!

  - Adjoint differentiation issues:
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs
     - or subgraphs!

- Adjoint differentiation issues:
  1. Needlessly expensive for many-output-variable, few-input variable graphs
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs
  - or subgraphs!

- Adjoint differentiation issues:
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  - or subgraphs!
Equation Derivatives

But *how well* is differentiation solved?

- Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs
     - or subgraphs!

- Adjoint differentiation issues:
  1. Needlessly expensive for many-output-variable, few-input variable graphs
     - or subgraphs!
  2. Typically requires additional temporary data
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  1. Needlessly expensive for many-input-variable, few-output variable graphs
  2. or subgraphs!

- **Adjoint differentiation issues:**
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  2. or subgraphs!
  3. Typically requires additional temporary data
     - particularly with operator overloading...
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  1. Needlessly expensive for many-input-variable, few-output variable graphs
  - or subgraphs!

- **Adjoint differentiation issues:**
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  - or subgraphs!
  2. Typically requires additional temporary data
  - particularly with operator overloading...
  3. No nice `DualNumber` representation?
Equation Derivatives

But *how well* is differentiation solved?

• Forward differentiation issues:
  1. Needlessly expensive for many-input-variable, few-output variable graphs
     • or subgraphs!

• Adjoint differentiation issues:
  1. Needlessly expensive for many-output-variable, few-input variable graphs
     • or subgraphs!
  2. Typically requires additional temporary data
     • particularly with operator overloading...
  3. No nice `DualNumber` representation?

• Optimal Jacobian Accumulation issues:
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  1. Needlessly expensive for many-input-variable, few-output variable graphs
     - or subgraphs!

- **Adjoint differentiation issues:**
  1. Needlessly expensive for many-output-variable, few-input variable graphs
     - or subgraphs!
  2. Typically requires additional temporary data
     - particularly with operator overloading...
  3. No nice `DualNumber` representation?

- **Optimal Jacobian Accumulation issues:**
  1. NP-complete!
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  1. Needlessly expensive for many-input-variable, few-output variable graphs
  2. or subgraphs!

- **Adjoint differentiation issues:**
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  2. or subgraphs!
  3. Typically requires additional temporary data
  4. particularly with operator overloading...
  5. No nice `DualNumber` representation?

- **Optimal Jacobian Accumulation issues:**
  1. NP-complete!
  2. don’t want to stress-test compilers...
Equation Derivatives

But *how well* is differentiation solved?

- **Forward differentiation issues:**
  1. Needlessly expensive for many-input-variable, few-output variable graphs
  2. or subgraphs!

- **Adjoint differentiation issues:**
  1. Needlessly expensive for many-output-variable, few-input variable graphs
  2. or subgraphs!
  3. Typically requires additional temporary data
     - particularly with operator overloading...
  4. No nice `DualNumber` representation?

- **Optimal Jacobian Accumulation issues:**
  1. NP-complete!
     - don’t want to stress-test compilers...

- **Solution:** “Meh, Good Enough” Jacobian Accumulation
Chain Rule Application: Condensing one Node

Each equation node can be used to define \textit{local} partial derivatives
Chain Rule Application: Condensing one Node

Each equation node can be used to define *local* partial derivatives

- $N_i$ inputs, $N_o$ outputs gives $N_i \cdot N_o$ partial derivatives
- Small equations are efficiently evaluated with sparse forward AD
- Equation nodes are out of the picture
- Partial derivatives are new directed edges with weights $D(V_i, V_j)$ (often 0) connecting variables $V_i$ and $V_j$
Chain Rule Application: Condensing one Node

Each equation node can be used to define *local* partial derivatives

- $N_i$ inputs, $N_o$ outputs gives $N_i \cdot N_o$ partial derivatives
- Small equations are efficiently evaluated with sparse forward AD
- Equation nodes are out of the picture
- Partial derivatives are new directed edges with weights $D(V_i, V_j)$ (often 0) connecting variables $V_i$ and $V_j$

An intermediate variable $M$ can be “condensed away”:
Chain Rule Application: Condensing one Node

Each equation node can be used to define local partial derivatives

- $N_i$ inputs, $N_o$ outputs gives $N_i \cdot N_o$ partial derivatives
- Small equations are efficiently evaluated with sparse forward AD
- Equation nodes are out of the picture
- Partial derivatives are new directed edges with weights $D(V_i, V_j)$ (often 0) connecting variables $V_i$ and $V_j$

An intermediate variable $M$ can be “condensed away”:

- For all outgoing edges to output variable $O$:
- For all incoming edges from input variable $I$:
  - $D(I, O) + = D(I, M) \cdot D(M, O)$
    - (creating a new edge if $D(I, O)$ was previously 0)
Chain Rule Application: Condensing one Node

Each equation node can be used to define *local* partial derivatives

- $N_i$ inputs, $N_o$ outputs gives $N_i \cdot N_o$ partial derivatives
- Small equations are efficiently evaluated with sparse forward AD
- Equation nodes are out of the picture
- Partial derivatives are new directed edges with weights $D(V_i, V_j)$ (often 0) connecting variables $V_i$ and $V_j$

An intermediate variable $M$ can be “condensed away”:

- For all outgoing edges to output variable $O$:
- For all incoming edges from input variable $I$:
  - $D(I, O) + = D(I, M) \cdot D(M, O)$
    - (creating a new edge if $D(I, O)$ was previously 0)

Just keep condensing until all remaining edges are directly from user inputs to user outputs
Chain Rule Application: Condensing the Graph

- In what order do we condense these equations?
  - $N$-factorial possibilities
  - Some possibilities are cheaper than others
  - $NP$-complete optimization problem
Chain Rule Application: Condensing the Graph

- Forward differentiation:
  - Evaluates equations inputs-to-outputs
  - Effectively does A, B, C, D, E, F
  - $3 + 3 + 3 + 6 + 9 + 9 = 33$ multiply+add operations
Chain Rule Application: Condensing the Graph

- Reverse differentiation:
  - Evaluates equations outputs-to-inputs
  - Effectively does F, E, D, C, B, A
  - Still $3 + 3 + 3 + 6 + 9 + 9 = 33$ multiply+add operations
Chain Rule Application: Condensing the Graph

- Greedy (lazy?) differentiation:
  - Pick cheapest condensations first
  - Does C, A, B, E, F, D
  - Still $2 + 3 + 3 + 3 + 3 + 9 = 23$ multiply+add operations: 30% cheaper!
Chain Rule Application: Condensing the Graph

Greedy (lazy?) differentiation:
- Pick cheapest condensations first
- Does C, A, B, E, F, D
- Still $2 + 3 + 3 + 3 + 3 + 9 = 23$ multiply+add operations: 30% cheaper!

Meh, good enough
Upcoming in Part 3

- Automatic Physics equation inversions
- Automatic Physics subgraph inversions
- Compile-time sparse matrix algorithm generation
  - Inversion
  - Factorization

Plus whatever else I think of while coding and writing part 3...