Generative Programming for Automatic Differentiation

Marco Nehmeier

Institute of Computer Science
University of Würzburg
Germany

AD 2012

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

5 Summary

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

5 Summary

My Background

- Computer science
- Interval arithmetic
  - Avoid rounding mode switches
  - Easy to use / DSL
- Implementation
  - C++ / C++11
  - Expression templates
  - Template meta programming

$\Rightarrow$ (Forward) Automatic differentiation

$\Rightarrow$ Prototype without interval arithmetic

C++ Templates
Generic Programming

- Generic classes
- Parametrized with
  - Types
    - typename
    - class
  - Values
    - int
    - bool
    - char
    - Enums
- Compile time
- Functions as well

```
template<typename T, int N>
class Container {
    T v[N];
public:
    ...
};

Container<string, 20> scon;
Container<double, 12> dcon;
```

```
template<typename T>
static T max(T m, T n) {
    return m > n ? m : n;
}

max(2,3);       // max<int>(2,3)
max(2.5,3.1);   // max<double>(2.5,3.1)
```

Traits

- Template specializations
- Maps types or values
- Associate at compile time
  - Information
  - Behavior

```
numeric_limits<int>::min();
numeric_limits<double>::max();
```

```
template<typename T>
struct Trait {
    static void f(T t) {
        cout << t * t << endl;
    }
};

template<typename T>
struct Trait<T*> {
    static void f(T* t) {
        cout << (*t) * (*t) << endl;
    }
};

template<typename T>
void f(T t) {
    Trait<T>::f(t);
}
```

Template meta programming

- Turing complete
- Recursive templates
- Template specializations
- Compile time
  - Computation
  - Generate specialized algorithms

```
template<int B,int N>
struct Pow {
    enum { val = B * Pow<B,N-1>::val };
};

template<int B>
struct Pow<B,0> {
    enum { val = 1 };
};


Pow<2,3>::value;     // 2^3
```

Template meta programming


```
// General case ( Cond == true )
template<bool Cond, typename Then, typename Else>
struct IF {
    typedef Then action;
};

// Cond == false
template<typename Then, typename Else>
struct IF<false, Then, Else> {
    typedef Else action;
};


IF<con, AlgoA, AlgoB>::action::execute();
```

Expression Templates

- Operator overloading

```
Vector v1, v2, v3;
...
Vector res = v1 + v2 + v3;
```

- Pairwise evaluation problem

```
...
Vector tmp = v1 + v2;
Vector res = tmp + v3;
```

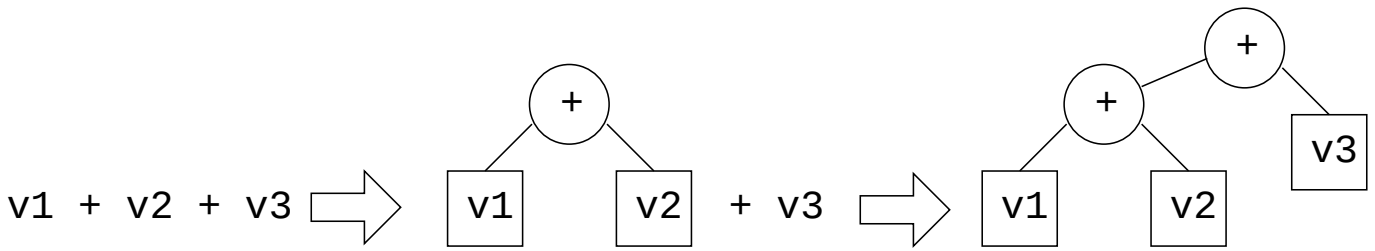
- Additional loops
- Additional instances
  - `new`
  - `delete`

Expression Templates

- Building parse trees
  - Operator overloading
  - Recursive templates
- Evaluate whole tree
  - Assignment
    - `operator=`
  - Optimized
    - Algorithm
    - Compiler

```
class V { };

struct Add { };

template<class L, class Op, class R>
struct X {
    V eval();
};

template<class T>
X<T, Add, V> operator+(T, V) {
    return X<T, Add, V>();
}
```

```
V res = v1 + v2 + v3;
   // = X<V,Add,V>() + v3;
   // = X<X<V,Add,V>,Add,V>().eval();
```



C++11
The new ISO C++ Standard

- New ISO C++ standard
  - Core language
    - Range-based for-loops
    - Lambda functions
    - Initializer lists
    - Uniform initialization
    - Delegating constructors
    - Extern templates
    - Alternative function syntax
    - Explicit virtual function overrides
    - User-defined literals
    - Multi threading
    - ...
  - Standard library
    - New container classes
    - Type traits
    - Regular expressions
    - ...

C++11
Eye-catching Features . . .

- Typedef template is not allowed

```
template<typename V> typedef map<string, V> sMap; // Illegal
```

- Template alias

```
template<typename V> using sMap = map<string, V>;

sMap<int> x; // map<string,int>
```

C++11
Eye-catching Features . . .

- Type inference
  - auto

```
auto x = some_function(...);
```

```
auto exp = v1 + v2;
      // = X<V,Add,V>();

V res = exp + v3;
   // = X<X<V,Add,V>,Add,V>().eval();
```

  - decltype

```
decltype(x + y) z = x + y;
```

```
template<class X, class Y, class Z>
auto fma(X x, Y y, Z z) -> decltype(x * y + z) {
    return x * y + z;
}

fma(5, 8, 2);     // 42
fma(v1, v2, v3); // X<X<V,Mul,V>,Add,V>
```

C++11
Eye-catching Features ...

- Variadic templates
  - Arbitrary number
  - Any type

```
template<class... V> class C;
```

- Split off

```
template<class F, class... V>
class C;
```

- Passing template arguments

```
typedef C<V...> c;
```

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

5 Summary

Symbolic differentiation

- J. Gil and Z. Gutterman 1998
- Expression templates
- Template meta programming

```
template<typename T1, typename T2> struct plus_t {
    typedef plus_t<typename T1::tag, typename T2::tag> tag;
};
```

$\Rightarrow$ Symbolic derivative `T::tag`

Forward automatic differentiation

- FAD 2001
- Expression templates

```
template <typename T> class Fad {
  protected:
    T val_;
    Vector<T> dx_;
    ...
};
```

- `diff(int ith, int n)`
- `val()`, `dx(int i)`

$\Rightarrow$ `TFad`

$\Rightarrow$ Sacado `DFad` and `SFad`

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

5 Summary

Main idea

- Expression tree
  - Symbolic characteristic
  - No functions like
    - `val()`
    - `dx(int i)`
    - `diff(int ith, int n)`
  - Variables and constants
    - `val()`

$\Rightarrow$ Record the structure of an expression

Main idea

- TMP classes / functions
- Traverse the tree at compile-time
  - Visitor pattern
- Transform tree into runtime-code

$\Rightarrow$ Extensibility
$\Rightarrow$ Replaceability
$\Rightarrow$ Efficiency

Example


```
Var<double, 'x'> x(1.5);
Var<double, 'y'> y(2.3);
Con<double> c(2.0);

// Create expression tree
auto expr = x * x + c * y;
// Perform automatic differentiation (implies code generation)
auto res = expr.df<2>();

// Shortform using res as an intermediate variable
auto res2 = ( sin(res) * y ).df<2>();

cout << "res  d/dx   " << get<'x', 1>(res)  << endl;
cout << "res2 d/dydy " << get<'y', 2>(res2) << endl;
```

Expression tree
Leaf nodes


```
template<typename T, unsigned int ID>
class Var {
    T val_;

  public:
    Var(T val) : val_(val) { } ;

    T val() const { return val_; }
};
```

- Underlying type `T`
- Identifier `ID`
  - `Var<double, 1>`
  - `Var<interval, 'x'>`

$\Rightarrow$ `Con<typename T>`

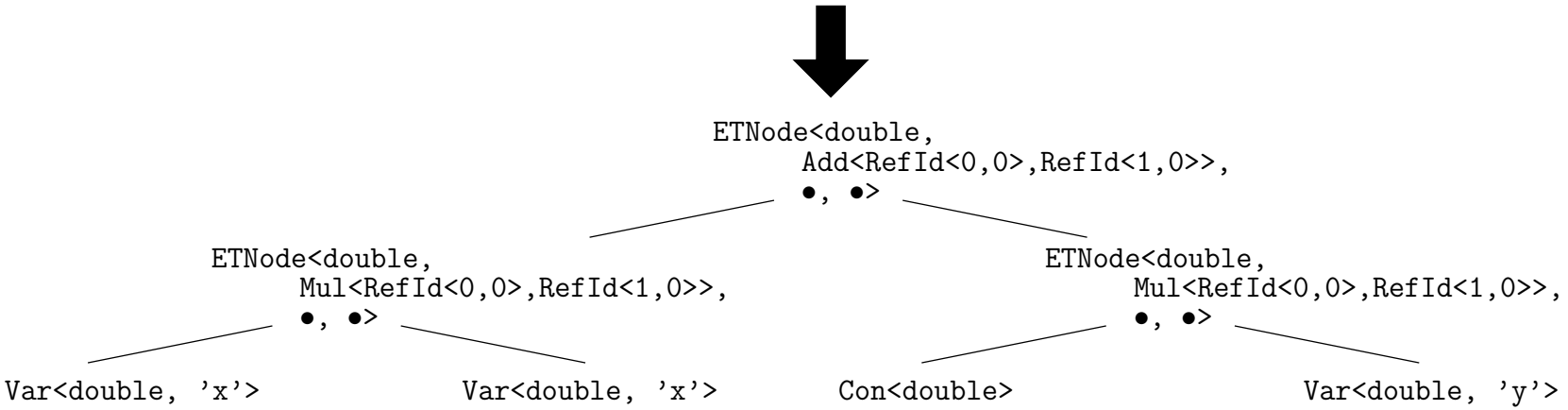
Expression tree
Inner nodes


```
template<typename T, typename Policy, typename... Para>
class ETNode {
  public:
    typedef Policy policy;
    typedef std::tuple<Para...> tuple;

    ETNode(Para const& ... para) : s_(para...) { }

    tuple& successor() { return s_; }
    tuple const& successor() const { return s_; }

  private:
    tuple s_;
};
```

- Arbitrary number of arguments
  - Accessible at compile-time
- Symbolic characteristic
  - Policy class

Expression tree
Structure


```
Var<double, 'x'> x(1.5);
Var<double, 'y'> y(2.3);
Con<double> c(2.0);

// Create expression tree
auto expr = x * x + c * y;
```



Policies


```
template<typename P>
struct Sin {
  // cos(P) * P'
  typedef Mul<Cos<P>, typename P::df> df;
};
```

- Symbolic behavior
- J. Gil and Z. Gutterman 1998
- Only on a single node

Policies


```
template<unsigned int I, unsigned int N>
struct RefId {
    typedef RefId<I, N + 1> df;
};
```

- Nth derivative
- Ith argument

```
Mul<RefId<0,0>,RefId<1,0>>::df
=Add<Mul<RefId<0,0>,RefId<1,1>>,
        Mul<RefId<0,1>,RefId<1,0>>>
```

Result type

- Derivative<T, N, VarList<Tail...>>

```
template<typename T, int N, int ID, int ... Tail>
class Derivative<T, N, VarList<ID, Tail... >>
  : public Derivative<T, N, VarList<Tail...>> {
    crTuple<N, T>::type t_;
  public:
    enum {var = ID};
    ...
};
```

```
template<typename T, int N>
class Derivative<T, N, VarList< >> {
  protected :
    T val_ ;
...
};
```

Code Generation
"Visitor class"

- `Eval<typename T, typename D, typename Node>`
- `static void eval(D& d, Node const& n)`
  1. Create instances $d_1, \ldots, d_n$ of the type `D`, stored in a tuple `t` of the type `std::tuple<D, ..., D>`.
  2. Use `Eval` recursively to evaluate the child nodes, writing their results into $d_1, \ldots, d_n$.
  3. Mix in the run-time code for the evaluation/differentiation of the independent variables $ID_1, \ldots, ID_k$ by applying the tuple `t` onto `EvalPol`.
     - `get<`ID_1`,0>(d) = EvalPol<T,`ID_1`,Policy>::eval(t)`
     - `get<`ID_1`,1>(d) = EvalPol<T,`ID_1`,Policy::df>::eval(t)`
     - `get<`ID_1`,2>(d) = EvalPol<T,`ID_1`,Policy::df::df>::eval(t)`
     - ...

Code Generation

- `EvalPol<typename T, int ID, typename Policy>`
  - Mapping
  - Symbolic policy class
  - Functions and operations
- `EvalPol<typename T, int ID, RefID<I, N>`
  - Termination condition
  - References of the child nodes

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

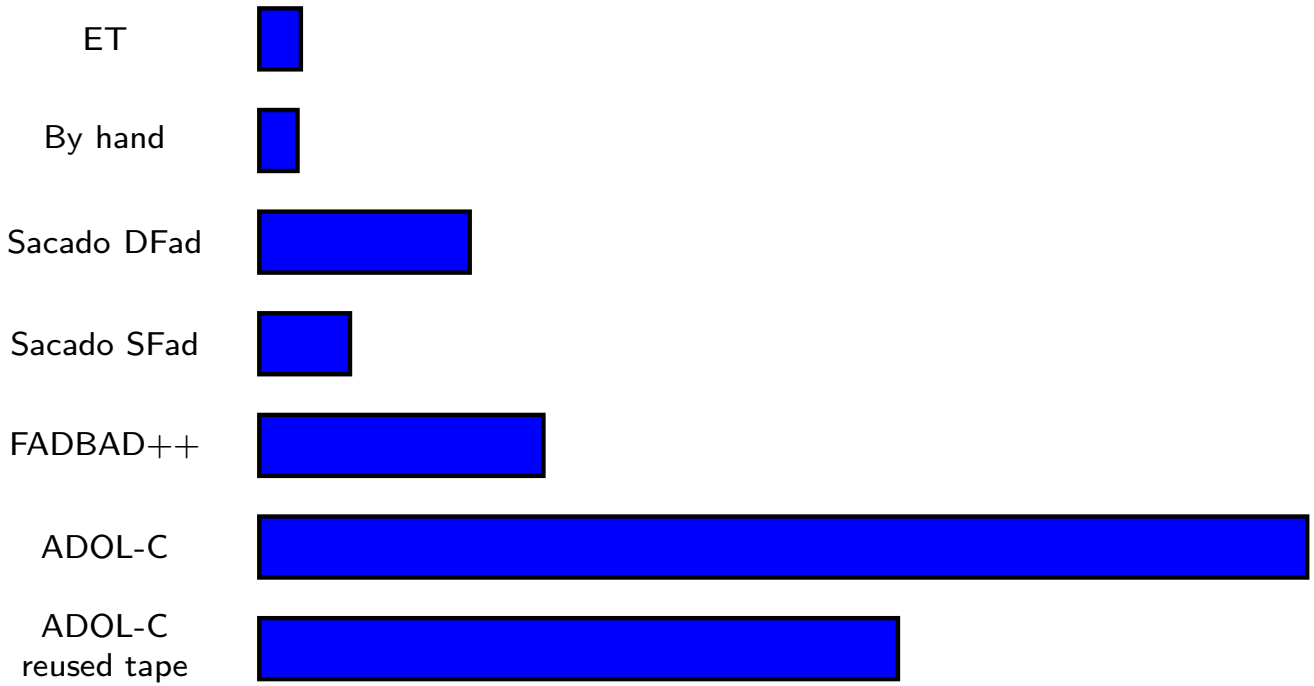
5 Summary

Test System

- Intel(R) Core(TM)2 Quad CPU Q9550@2.83GHz
- 8 GB RAM
- Linux 3.0.0 64 Bit
- GNU g++ 4.4.6
  - `-O2`
  - `-std=c++0x`
- FADBAD++ 2.1
- ADOL-C 2.2.0
- Trilinos Sacado 10.4
- filib++ 3.0.2
- C-XSC 2.4.0

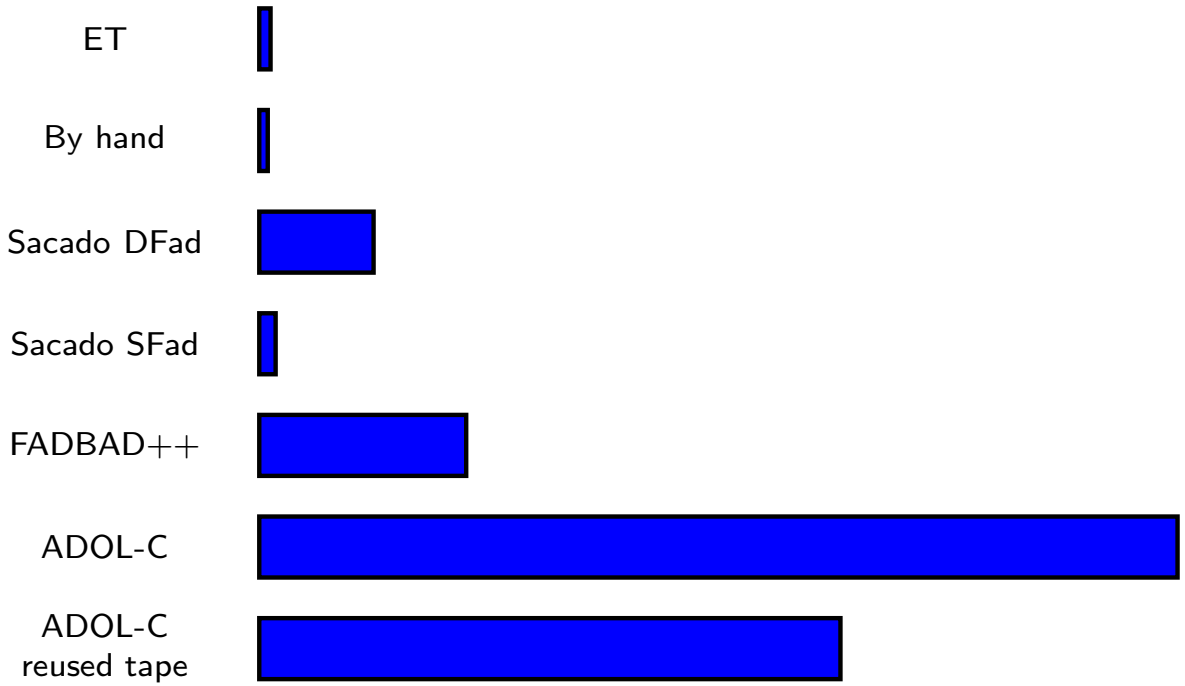
Simple functions
double

$$x^2y^3 + y\log(x)$$



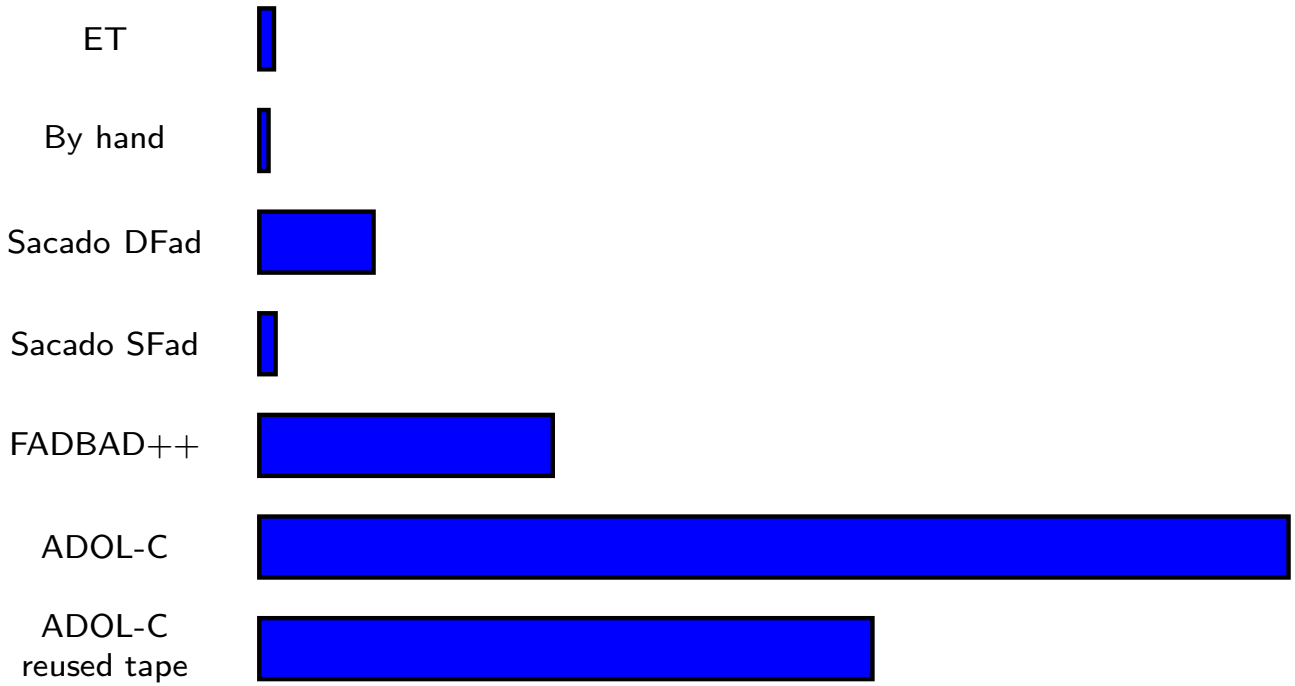
Simple functions
double

$$3x^2y - y^3$$



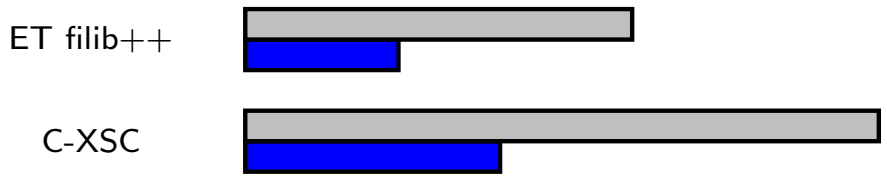
Simple functions
doubles

$$(1 - x)^2 + 100(y - x^2)^2$$



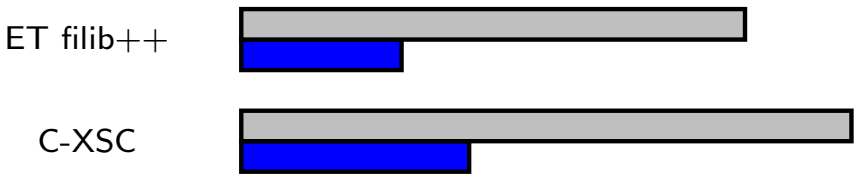
Simple functions
Interval

$$x^3 + x - 1$$



Simple functions
Interval

$$e^x \sin(4x)$$



Interval Halley's method

$$a^{(k)} := -\frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

$$b^{(k)} := a^{(k)} \cdot \frac{f''\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

$$x^{(k+1)} := x^{(k)} + \frac{a^{(k)}}{1 + \frac{b^{(k)}}{2}}$$

- Speed up
  - $x^3 + x - 1$: 1.76
  - $e^x \sin(4x)$: 1.39

Outline

1 Introduction

2 Related Work

3 Implementation

4 Experimental Results

5 Summary

Summary

$\Rightarrow$ Extensibility
$\Rightarrow$ Replaceability
$\Rightarrow$ Efficiency

Questions ?

Marco Nehmeier

nehmeier@informatik.uni-wuerzburg.de