Interval Arithmetic using Expression Templates, Template Meta Programming and the upcoming C++ Standard

Marco Nehmeier

Institute of Computer Science
University of Würzburg
Germany

SCAN 2010

Outline

1 Introduction

2 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

Outline

1 Introduction

2 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

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 { value =
        Base * Pow<B,N-1>::value };
};

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

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

Expression Templates

- Operator overloading

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

- Pairwise evaluation problem

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

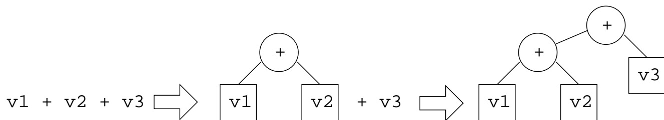
- Additional loops
- Additional instances
  - `new`
  - `delete`

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();
```



Outline

1 Introduction

2 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

C++0x

The upcoming ISO C++ Standard

- Upcoming ISO C++ standard
- Draft was published in August 2010

- 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
  - ...

- 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>
```

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>
```

- 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 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

Former Work

Interval arithmetic using expression templates

- Reset rounding mode
- Pairwise evaluation problem
- Unnecessary operations

```
Interval a, b, c;

Interval r = a + b + c;
```

```
store_rounding();
set_rounding_downward();
t_inf = a_inf + b_inf;
t_sup = -(-a_sup - b_sup);
reset_rounding();

store_rounding();
set_rounding_downward();
r_inf = t_inf + c_inf;
r_sup = -(-t_sup - c_sup);
reset_rounding();
```

- Direct rounding not always possible!

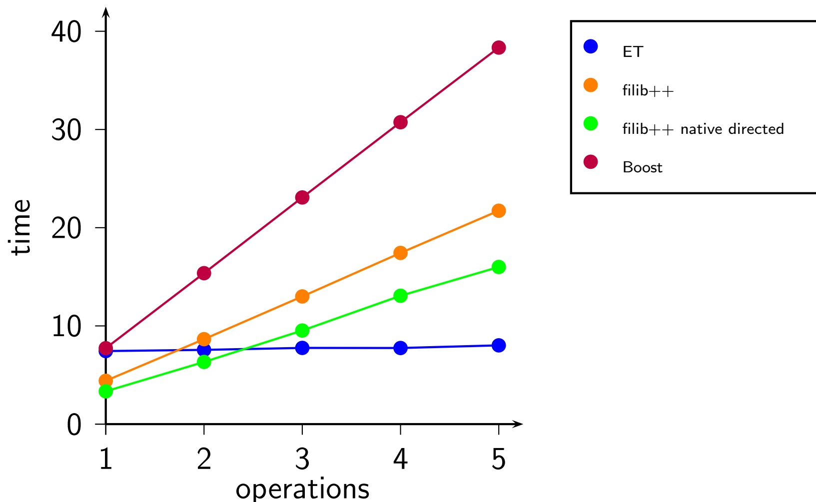
Former Work

Interval arithmetic using expression templates

- Expression templates
- Building parse tree
- Evaluate whole tree
  1. Create `RndControl` instance
     - Store initial rounding
  2. Pass `RndControl` instance through the tree
     - Manage rounding mode switches
  3. Destroy `RndControl` instance
     - Reset to initial rounding

Former Work

Interval arithmetic using expression templates



Outline

1 Introduction

2 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

Improvements

- Rounding mode switches
- Flexible tree structure
- Automatic differentiation

$\Rightarrow$ Template meta programming

Optimize Rounding Mode Switches

- Drawback of `RndControl`
  - Works at runtime
    - Visitor
    - Branches

$\Rightarrow$ Could be done during compile time!

Optimize Rounding Mode Switches

- Drawback of `RndControl`
  - Works at runtime
    - Visitor
    - Branches

⇒ Could be done during compile time!

Optimize Rounding Mode Switches

- Operation "knows" rounding
  - Required
  - Outcoming

```
class I { ... };

template<class L, class Op, class R>
struct X { ... };
```

```
enum class RndState : int {
    UNKNOWN,
    UNCHANGED,
    DOWNWARD,
    ...
};
```

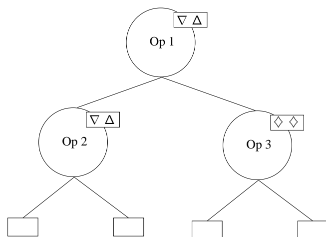
```
struct Add {
    static const RndState rndB = ...
    static const RndState rndA = ...

    static I eval(I a, I b) { ... }
};
```

Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. Eval<T,UNKNOWN>
  3. Reset rounding

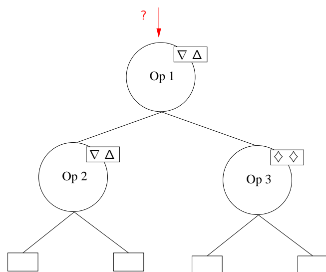
- Eval<X<L,Op,R>,Rnd>
  1. Eval<L,Rnd>
  2. Eval<R,EL::rnd>
  3. Set rounding to Op::rndB
  4. Op::eval(l,r)
  5. rnd = Op::rndA

- Eval<I,Rnd>
  - UNCHANGED



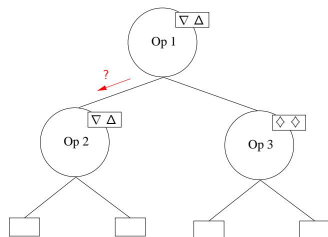
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



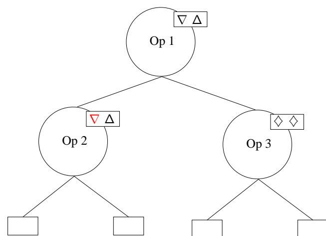
Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. Eval<T,UNKNOWN>
  3. Reset rounding
- Eval<X<L,Op,R>,Rnd>
  1. Eval<L,Rnd>
  2. Eval<R,EL::rnd>
  3. Set rounding to Op::rndB
  4. Op::eval(l,r)
  5. rnd = Op::rndA
- Eval<I,Rnd>
  - UNCHANGED



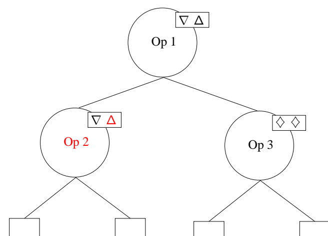
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



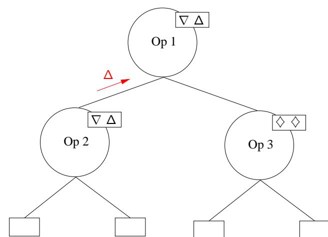
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



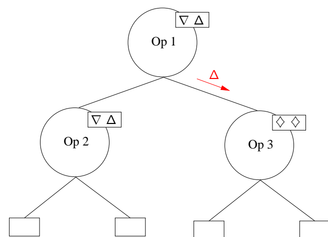
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



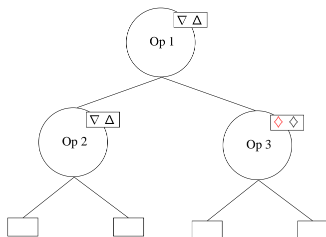
Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



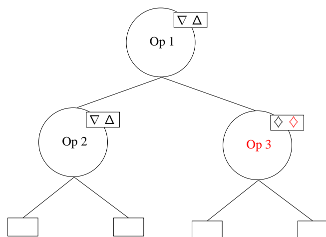
Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. Eval<T,UNKNOWN>
  3. Reset rounding
- Eval<X<L,Op,R>,Rnd>
  1. Eval<L,Rnd>
  2. Eval<R,EL::rnd>
  3. Set rounding to Op::rndB
  4. Op::eval(l,r)
  5. rnd = Op::rndA
- Eval<I,Rnd>
  - UNCHANGED



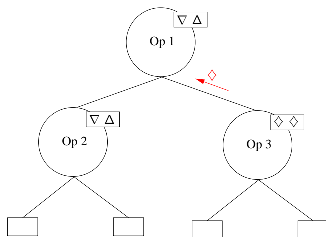
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



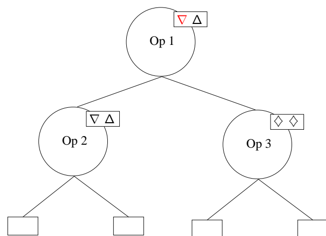
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



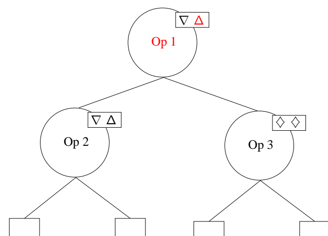
Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. Eval<T,UNKNOWN>
  3. Reset rounding
- Eval<X<L,Op,R>,Rnd>
  1. Eval<L,Rnd>
  2. Eval<R,EL::rnd>
  3. Set rounding to Op::rndB
  4. Op::eval(l,r)
  5. rnd = Op::rndA
- Eval<I,Rnd>
  - UNCHANGED

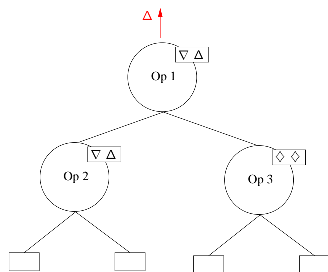


Optimize Rounding Mode Switches

- Evaluation of tree T
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding

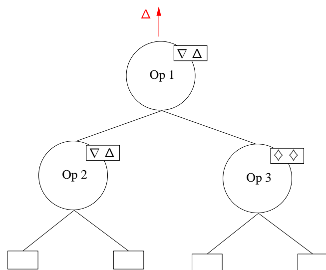
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`

- `Eval<I,Rnd>`
  - UNCHANGED



Optimize Rounding Mode Switches

- Evaluation of tree `T`
  1. Store initial rounding
  2. `Eval<T,UNKNOWN>`
  3. Reset rounding
- `Eval<X<L,Op,R>,Rnd>`
  1. `Eval<L,Rnd>`
  2. `Eval<R,EL::rnd>`
  3. Set rounding to `Op::rndB`
  4. `Op::eval(l,r)`
  5. `rnd = Op::rndA`
- `Eval<I,Rnd>`
  - `UNCHANGED`



Optimize Rounding Mode Switches

- Switching rounding mode
- Traits
- Not as easy...
  - UNCHANGED
  - UNKNOWN
    - Necessary?

```
template<RndState OLD, RndState NEW>
struct RndSet {
    static void set() {
        /* set Rounding mode */
        ...
    }
};

template<RndState RND>
struct RndSet<RND, RND> {
    // Dummy
    static void set() { }
};

...
```

Flexible Tree Structure

- Parse tree
  - Up to now
    - Binary
    - `X<L,Op,R>`
  - Unary, ternary, ...?
    - Functions
    - Reverse operations
  - New expression classes
  - New traits specializations

⇒ Variadic templates!

Flexible Tree Structure

- Parse tree
  - Up to now
    - Binary
    - `X<L,Op,R>`
  - Unary, ternary, ...?
    - Functions
    - Reverse operations
  - New expression classes
  - New traits specializations

⇒ Variadic templates!

Flexible Tree Structure

- Variadic tuple
  - Split off
    - Head `H`
    - Tail `T...`
  - Derive `Tuple<T...>`
  - Store head
  - Getter
    - Head
    - Tail
- `EAt<N,Tuple<H,T...>>`
  - Recursive call
    `EAt<N-1,Tuple<T...>>`
- `EAt<0,Tuple<H,T...>>`
  - Return head

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

template<> class Tuple<> {};

template<class H, class... T>
class Tuple : private Tuple<T...> {
    H head_;
public:
    Tuple() {}

    Tuple(H const& h, T const& ... t)
        : Tuple<T...>, head_(h) {}

    H& head() { return head_; }

    Tuple<T...>& tail() {
        return *this;
    }
};
```

```
Tuple<int, double, int> t(1, 2.1, 3);

EAt<1,decltype(t)>::get(t);    // 2.1
```

Flexible Tree Structure

- Expression class `X<Op,Succ...>`
  - Policy
    - `class Op`
  - Successors
    - `class... Succ`
  - Derive from `Tuple<Succ...>`
- Modify evaluation traits
  1. Create `Tuple<I,...,I>` of size `Succ`
  2. Evaluate successors writing results into tuple
  3. Apply tuple to policy

⇒ Easy to use/extend

⇒ Policy

⇒ Operator/function

Flexible Tree Structure

- Expression class `X<Op,Succ...>`
  - Policy
    - `class Op`
  - Successors
    - `class... Succ`
  - Derive from `Tuple<Succ...>`
- Modify evaluation traits
  1. Create `Tuple<I,...,I>` of size `Succ`
  2. Evaluate successors writing results into tuple
  3. Apply tuple to policy

⇒ Easy to use/extend
  ⇒ Policy
  ⇒ Operator/function

Automatic Differentiation

- Symbolic differentiation
  - Computes formal expression
  - Slow
- Automatic differentiation
  - Compute expression and derivative
    - $(v, v') + (u, u') = (v + u, v' + u')$
  - Two common approaches
    - Pre processing
    - Operator overloading
  - Inflexible
- Numerical differentiation
  - Imprecise

$\Rightarrow$ Combine symbolic and automatic differentiation

Automatic Differentiation

- Symbolic differentiation
  - Computes formal expression
  - Slow
- Automatic differentiation
  - Compute expression and derivative
    - $(v, v') + (u, u') = (v + u, v' + u')$
  - Two common approaches
    - Pre processing
    - Operator overloading
  - Inflexible
- Numerical differentiation
  - Imprecise

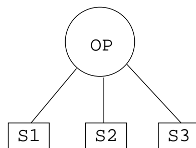
$\Rightarrow$ Combine symbolic and automatic differentiation

Automatic Differentiation

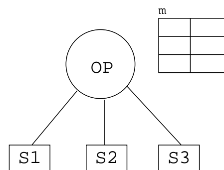
- New leaf types
  - `Const`
  - `Var`
- Modify evaluation traits
  - Number `N` of max. derivative
  - Return a `Tuple<I,...,I>` of size $\mathtt{N}+1$
  - Specializations
    - `Const`: $[\mathtt{I}, \underbrace{0,\ldots,0}_{\mathtt{N}}]$
    - `Var`: $[\mathtt{I}, 1, \underbrace{0,\ldots,0}_{\mathtt{N}-1}]$

Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

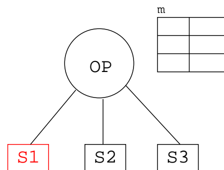


- Modify evaluation traits for X<Op,Succ...>
  1. Create P × (N + 1) tuple matrix m of type I
  2. Evaluate successors writing result tuples into matrix m
  3. Create N + 1 tuple t of type I
  4. Apply matrix m to policy Op
     - EAt<0,T>::get(t) = EvalPol<Op>::eval(m)
     - EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)
     - ...
  5. Return tuple t

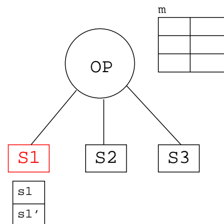


Automatic Differentiation

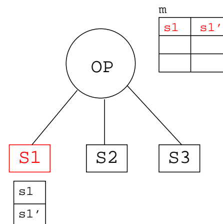
- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

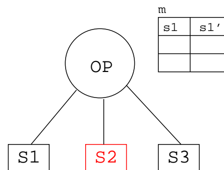


- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

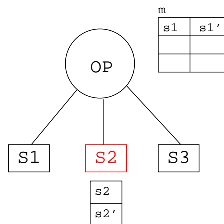


Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

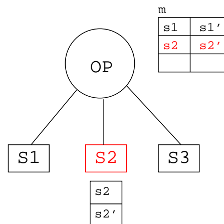


- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

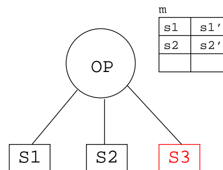


Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

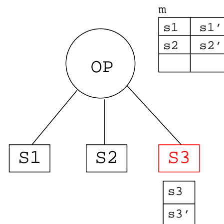


- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



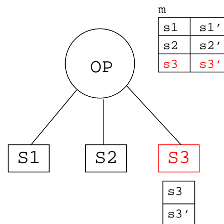
Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create `P` $\times$ (`N` + 1) tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create `N` + 1 tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



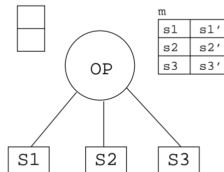
Automatic Differentiation

- Modify evaluation traits for X<Op,Succ...>
  1. Create $P \times (N+1)$ tuple matrix m of type I
  2. Evaluate successors writing result tuples into matrix m
  3. Create $N+1$ tuple t of type I
  4. Apply matrix m to policy Op
     - EAt<0,T>::get(t) = EvalPol<Op>::eval(m)
     - EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)
     - ...
  5. Return tuple t



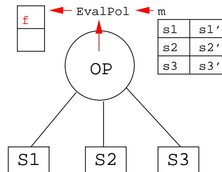
Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $\mathtt{P} \times (\mathtt{N}+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $\mathtt{N}+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



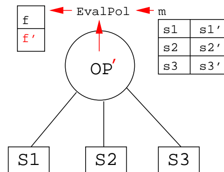
Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $P \times (N+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $N+1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`

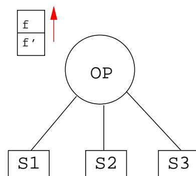


Automatic Differentiation

- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $P \times (N+1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $N + 1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



- Modify evaluation traits for `X<Op,Succ...>`
  1. Create $P \times (N + 1)$ tuple matrix `m` of type `I`
  2. Evaluate successors writing result tuples into matrix `m`
  3. Create $N + 1$ tuple `t` of type `I`
  4. Apply matrix `m` to policy `Op`
     - `EAt<0,T>::get(t) = EvalPol<Op>::eval(m)`
     - `EAt<1,T>::get(t) = EvalPol<Op::df>::eval(m)`
     - ...
  5. Return tuple `t`



Automatic Differentiation

- New policy classes
  - Annotation
- Replace methods by traits
  - EvalPol<Op>

```
template<size_t P, size_t N>
struct Df {
    typedef Df<P,N+1> df;
};
```

```
template<class S1, class S2>
struct Add {
    typedef Add<typename S1::df, typename S2::df> df;
};
```

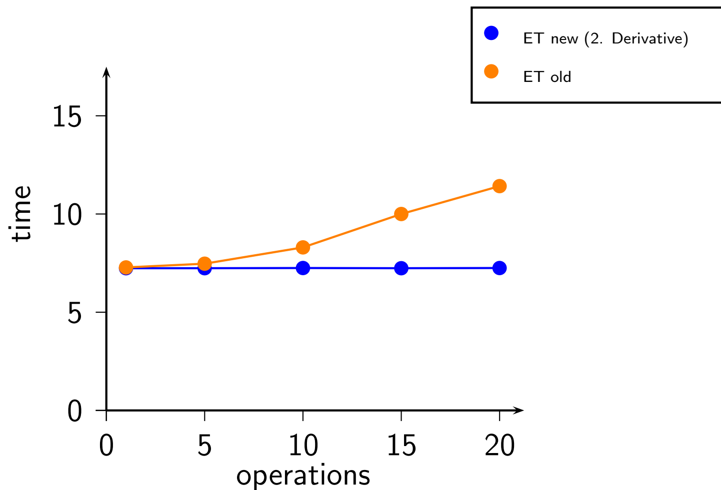
- Op = Add<Df<0,0>,Df<1,0>>
- Op’ = Op::df = Add<Df<0,1>,Df<1,1>>

Domain specific language

- Domain specific language
- `Derivative<N>`
- Assignment
  - Evaluate tree
  - Rounding
  - Automatic differentiation
- Interval type as well

```
typedef Derivative<3> DF;

// DSL for differentiation
auto expr = I(0.5,3) * Var() + ...;

DF res = expr(I(1.0,2.5));

// only constants
I x = I(0.5,3) + I(1.0,2.5);
```

Runtime



Outline

1 Introduction

2 C++0x

3 Interval Arithmetic using Expression Templates

4 Improvements

5 Summary

Summary

- Flexible
  - Easy to extend
  - Automatic differentiation
- Easy to use
  - DSL
- Fast

Questions ?

Marco Nehmeier

nehmeier@informatik.uni-wuerzburg.de