

Metaprogramming for math library

Jiří Chmelík, [Marek Trtík](#)
PA199

Metaprogramming

- ▶ **Metaprogram** is a code whose **compilation** generates a **new code** whose functionality we really want.
- ▶ First metaprogram:
 - ▶ Erwin Unruh, 1994.
 - ▶ During compilation of his program the **compiler computed** prime numbers.
- ▶ Our first metaprogram will be much simpler...

Our first metaprogram

▶ Let's make a C++ compiler to compute 3^4 during compilation.

▶ Our metaprogram:

```
template< int N > struct Pow3 {
    enum { result = 3 * Pow3<N-1>::result }; // Recursion rule
};

template< struct Pow3<0> {
    enum { result = 1 }; // Base case
};

int main() {
    std::cout << Pow3<4>::result;
}
```

▶ Resulting program:

```
std::cout << 3*3*3*3*1;
```

Unrolling loops: Motivation

- ▶ Let's consider this implementation of the dot product:

```
float dot_product(int dim, float* u, float* v) {  
    float result = 0.0f;  
    for (int i=0; i<dim; ++i)  
        result += u[i] * v[i];  
    return result;  
}
```

```
float u[3] = {1,2,3};    // A 3D vector
```

```
float v[3] = {4,5,6};    // A 3D vector
```

```
std::cout << dot_product(3, u, v);    // Print their dot product.
```

- ▶ Faster code would be: `std::cout << u[0]*v[0] + u[1]*v[1] + u[2]*v[2];`

Unrolling loops: Solution

```
template<int DIM> struct DotProduct { // Recursion rule
    static inline float result(float* u, float* v) {
        return (*u) * (*v) + DotProduct<DIM-1>::result(u+1, v+1);
    }
};
```

```
template<> struct DotProduct<1> { // Base case
    static inline float result(float* u, float* v) {
        return (*u) * (*v);
    }
};
```

Unrolling loops: Solution

```
template<int DIM> inline float dot_product(float* u, float* v) {  
    return DotProduct<DIM>::result(u, v);  
}
```

- ▶ The compiler then converts the call

`dot_product<3>(u, v)`

into code:

- ▶ `DotProduct<3>::result(u, v);`
- ▶ `(*u) * (*v) + DotProduct<2>::result(u+1, v+1);`
- ▶ `(*u) * (*v) + ((*u+1) * (*v+1) + DotProduct<1>::result(u+2, v+2));`
- ▶ `(*u) * (*v) + ((*u+1) * (*v+1) + (*u+2) * (*v+2));`

Expression templates: Motivation

- ▶ When designing a math library, it is desirable to provide this syntax:

```
Vector u(1000), v(1000);    // Define vectors
...                          // Initialise them
u = 1.2f * u + u * v;      // Perform operations.
```

- ▶ In C++ we can achieve this syntax via **operator overloading**.
- ▶ Question: Could there be a performance issue with such approach?
- ▶ Let's start with a naïve solution...

Expression templates: Motivation

```
class SVector { // "Simple" vector of any dimension.
    int size_;
    float* data_;
    void copy(SVector const& o) { for (int i=0; i<size(); ++i) data_[i] = o.data_[i]; }
public:
    explicit SVector(int size) : size_(size), data_(new float[size_]) {}
    ~SVector() { delete [] data_; }
    SVector(SVector const& o) : size_(o.size()), data_(new float[size_]) { copy(o); }
    SVector& operator=(SVector const& o) { copy(o); return *this; }
    int size() const { return size_; }
    float operator[](int const i) const { return data_[i]; }
    float& operator[](int const i) { return data_[i]; }
};
```


Expression templates: Motivation

```
SVector operator+(SVector const& u, SVector const& v) {  
    SVector result(u.size()); // A temporary for result computation.  
    for (int i=0; i<u.size(); ++i)  
        result[i] = u[i] + v[i];  
    return result;  
}
```

```
SVector operator*(SVector const& u, SVector const& v) {  
    SVector result(u.size()); // A temporary for result computation.  
    for (int i=0; i<u.size(); ++i)  
        result[i] = u[i] * v[i];  
    return result;  
}
```

Expression templates: Motivation

```
SVector operator*(float const a, SVector const& u) {  
    SVector result(u.size()); // A temporary for result computation.  
    for (int i=0; i<u.size(); ++i)  
        result[i] = a * u[i];  
    return result;  
}
```

Expression templates: Motivation

- ▶ Let's now use the code:

```
SVector u(1000), v(1000);  
u = 1.2f * u + u * v;
```

- ▶ How efficient is this code?

- ▶ `tmp1 = 1.2f * u;` // `tmp1` == “result” variable in `operator*`.
 - ▶ `operator*` creates “result” (allocation of “data_”). Then 1000 iterations in the loop.
- ▶ `tmp2 = u * v;` // `tmp2` == “result” variable in `operator*`.
 - ▶ `operator*` creates “result” (allocation of “data_”). Then 1000 iterations in the loop.
- ▶ `tmp3 = tmp1 + tmp2;` // `tmp3` == “result” variable in `operator+`.
 - ▶ `operator+` creates “result” (allocation of “data_”). Then 1000 iterations in the loop.
- ▶ `u = tmp3;`
 - ▶ `operator=` performs 1000 iterations in the loop.
 - ▶ Then 3 deallocations of arrays “data_” in `tmp1`, `tmp2`, and `tmp3`.

Expression templates: Motivation

- ▶ We can certainly do better (optimal code):

```
for (int i=0; i<u.size(); ++i)
    u[i] = 1.2f * u[i] + u[i] * v[i]; // Temporaries are CPU registers!
```

- ▶ How efficient is this code?

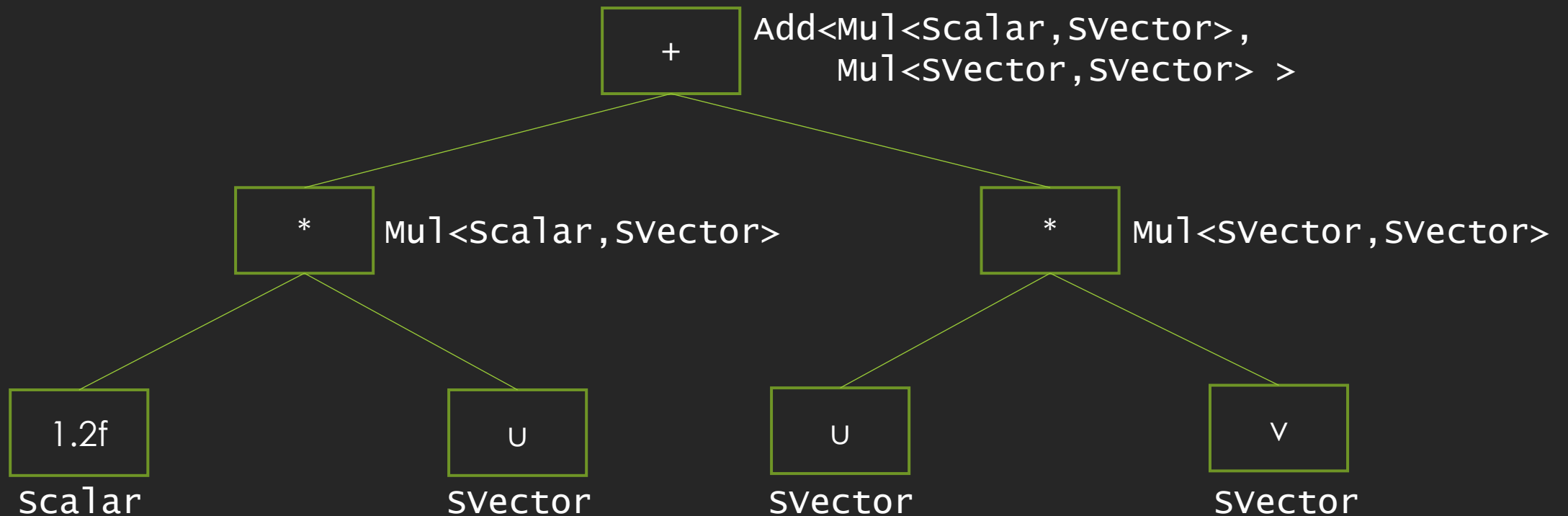
- ▶ **NO RAM TEMPORARIES!** (No allocations, no redundant RAM loads/stores).
- ▶ Just 1000 iterations in the loop.

- ▶ Can we achieve this efficiency while preserving the syntax?

- ▶ YES – using **EXPRESSION TEMPLATES**.

Expression templates: Idea

- ▶ We let the compiler to transform the **syntax tree** of the expression “`u = 1.2f * u + u * v`” by **hierarchy of simple template classes**:



Expression templates: Scalar

```
class scalar {  
    float const& s_;  
public:  
    scalar(float const& s) : s_(s) {}  
    int size() const { return 0; }  
    float operator[](int const) const { return s_; }  
};
```

Expression templates: ArgType

- ▶ Inside operator classes Add and Mul we must distinguish how we reference operands:
 - ▶ Scalar operand – by value.
 - ▶ Other operands – by reference.

```
template<typename T> struct ArgType {  
    typedef T const& result;  
};  
template<> struct ArgType<Scalar> {  
    typedef Scalar result;  
};
```

Expression templates: Mul

```
template<typename OP1, typename OP2>
class Mul {
    typename ArgType<OP1>::result op1_;
    typename ArgType<OP2>::result op2_;
public:
    Mul(OP1 const& op1, OP2 const& op2) : op1_(op1), op2_(op2) {}
    int size() const { return op1.size() != 0 ? op1.size() : op2.size(); }
    float operator[](int const i) const { return op1_[i] * op2_[i]; }
};
```


Expression templates: Add

```
template<typename OP1, typename OP2>
class Add {
    typename ArgType<OP1>::result op1_;
    typename ArgType<OP2>::result op2_;
public:
    Add(OP1 const& op1, OP2 const& op2) : op1_(op1), op2_(op2) {}
    int size() const { return op1.size() != 0 ? op1.size() : op2.size(); }
    float operator[](int const i) const { return op1_[i] + op2_[i]; }
};
```

Expression templates: Expression class

- ▶ How do we define operators? How about this:

```
template<typename T1, typename T2>  
Mul<T1,T2> operator*(T1 const& u, T2 const& v) { ... } // (1)
```

- ▶ Not good, parameters are too general, e.g.:

```
struct A {} a;  
struct B : public A {} b;  
template<typename T>  
A operator*(T const& s, A const& a) { ... } // (2)  
3 * a; // Always calls (2)  
3 * b; // calls (2) only when (1) is not defined.
```

Expression templates: Expression class

- ▶ We also do not want define all possible versions (too much work):

```
Mul< SVector, SVector > operator*(SVector const& u, SVector const& v);  
template<typename T> Mul<SVector,Mul<T> > operator*(SVector const& u, Mul<SVector,T> const& v);  
template<typename T> Mul<Mul<T>, SVector> operator*(Mul<SVector,T> const& u, SVector const& v);  
template<typename T> Mul< SVector,Add<T> > operator*(SVector const& u, Add<SVector,T> const& v);
```

...

- ▶ Solution: We introduce a class “Expr” for **wrapping** our expressions:

```
template<typename Rep> class Expr;
```

- ▶ We can safely declare the operator as:

```
template<typename OP1, typename OP2>  
Expr<Mul<OP1,OP2> > operator*(  
    Expr<OP1> const& u, Expr<OP2> const& v);
```

Expression templates: Expression

```
template<typename Rep>
class Expr { // Expression
    Rep rep_;
public:
    explicit Expr(int size) : rep_(size) {}
    Expr(Rep const& rep) : rep_(rep) {}
    Rep const& rep() const { return rep_; } // Unwrap the instance.
    int size() const { return rep_.size(); }
    float operator[](int const i) const { return rep_[i]; }
    // Defined later:
    template<typename Rep2> Expr& operator=(Expr<Rep2> const& o);
};
```

Expression templates: Operators

```
template<typename OP1, typename OP2>
Expr<Mul<OP1,OP2> > operator*(Expr<OP1> const& u, Expr<OP2> const& v) {
    return Expr<Mul<OP1,OP2> >(Mul<OP1, OP2>(u.rep(), v.rep()));
}
```

Wrap the result Actual operation Unwrap operands

```
template<typename OP2>
Expr<Mul<Scalar,OP2> > operator*(float const& s, Expr<OP2> const& u) {
    return Expr<Mul<Scalar,OP2> >(Mul<Scalar, OP2>(Scalar(s), u.rep()));
}
```

```
template<typename OP1, typename OP2>
Expr<Add<OP1,OP2> > operator+(Expr<OP1> const& u, Expr<OP2> const& v) {
    return Expr<Add<OP1,OP2> >(Add<OP1, OP2>(u.rep(), v.rep()));
}
```

Expression templates: Expr::operator=

```
template<typename Rep>
template<typename Rep2>
Expr<Rep>& Expr<Rep>::operator=(Expr<Rep2> const& o) {
    for (int i=0; i<size(); ++i)
        rep_[i] = o[i];
    return *this;
}
```

Add<Mul<Scalar, SVector>, Mul<SVector, SVector> >

Compiler converts 'o[i]' to
'1.2f*u[i] + u[i]*v[i]'
=> we get the efficient version!

► o[i] ==> Add[i] (Expr<Add<...> > forwards operator[] to 'rep')

==> Add::op1[i] + Add::op2[i]

==> Mul::op1[i] * Mul::op2[i] + Mul::op1[i] * Mul::op2[i]

==> Scalar[i] * SVector[i] + SVector[i] * SVector[i]

==> 1.2f * u[i] + u[i] * v[i]

Expression templates: Notes & Limitations

- ▶ We must declare our vector variables as:

```
Expr<SVector> u(1000), v(1000);    // Not nice!
```

- ▶ Therefore, we usually define:

```
typedef Expr<SVector> Vector;
```

- ▶ Limitation: Does not work for operations where RAM temporaries are really needed, e.g., ' $x = A*x$ ', where x is a vector and A is a matrix.
 - ▶ But ' $y = A*x$ ' would work just fine (assuming x, y are **different** vectors).
- ▶ Usage rule of thumb:
 - ▶ For **small fixed-size** vectors use **metaprogramming**, e.g., unrolling loops.
 - ▶ For **large dynamic-size** vectors use **expression templates**.

References

[1] D.Vandevoorde, N.M.Josuttis; *C++ Templates, The Complete Guide*; Addison-Wesley, 2006.