Expression Templates for Efficient, Generic Finance Code

Bowie Owens

2019-09-18

Outline

- Legacy structure and motivation for change.
- Expression templates in relationship to operator overloading.
- Using C++17 to implement expression templates.
- Concluding remarks.

Legacy structure and motivation for change.

Legacy code

```
TridiagScalarSum4(column,kappaLocal[tau]*thetaLocal[tau],temp1P
lusOneLogvol,temp1MidLogvol,temp1MinusOneLogvol,-
1*kappaLocal[tau]-
0.5*(lambdaLocal[tau]*etaLocal[tau])*(lambdaLocal[tau]*etaLocal
[tau]),plusOneSeriesFDFirstDevLogvol,midSeriesFDFirstDevLogvol,
minusOneSeriesFDFirstDevLogvol,0.5*(lambdaLocal[tau]*etaLocal[t
au])*(lambdaLocal[tau]*etaLocal[tau]),plusOneSeriesFDSecondDevL
ogvol,midSeriesFDSecondDevLogvol,minusOneSeriesFDSecondDevLogvo
l,-
0.5*yieldDomesticSeries[tau],zeroVectorLogvol,onesLogvol,zeroVe
ctorLogvol,temp1PlusOneLogvol,temp1MidLogvol,temp1MinusOneLogvo
l);
```

Operator Overloading for Clarity

```
class tridiagonal {
    std::vector<double> v_;
    // ...
};
tridiagonal operator+(
    tridiagonal const& lhs, tridiagonal const& rhs);
tridiagonal operator*(
    double lhs, tridiagonal const& rhs);
```

We’re Going to Need Tests…

- … lots of tests.
- When refactoring we want to preserve the behaviour of the software.
- Quickly testing legacy code - Clare Macrae [C++ on Sea 2019]

- https://www.youtube.com/watch?v=dtm8V3TIB6k

With Operator Overloading

```
temp1Logvol =
    (kappaLocal[tau] * thetaLocal[tau]) *
        temp1Logvol +
    (-1 * kappaLocal[tau] -
     0.5 * (lambdaLocal[tau] * etaLocal[tau]) *
           (lambdaLocal[tau] * etaLocal[tau])) *
        SeriesFDFirstDevLogvol +
    (0.5 * (lambdaLocal[tau] * etaLocal[tau]) *
           (lambdaLocal[tau] * etaLocal[tau])) *
        SeriesFDSecondDevLogvol +
    (-0.5 * yieldDomesticSeries[tau]) * ILogvol;
```

An Equivalent Example

```
double a, b;
tridiagonal A, B;
// …
A =
    a * A +
    b * B;
```

Temporaries (intermediate values)

```
// calculations done here
tridiagonal x1 = a * A;
tridiagonal x2 = b * B;
tridiagonal x3 = std::move(x1) + std::move(x2);

// no calculations done here
A = std::move(x3);

// for n=400, roughly 12x slower than TridiagScalarSum4()
```

Expression templates in relationship to operator overload ing.

Operator Overloading With Expression Templates

```
using add = /* … */;
using mul = /* … */;

template <class callable, class... operands>
class expr { /*...*/ };


expr<add, tridiagonal, tridiagonal>
operator+(
  tridiagonal const& a, tridiagonal const& b);


expr<mul, double, tridiagonal>
operator*(double a, tridiagonal const& b);
```

Operator Overloading With Expression Templates

```
using add = /* … */;
using mul = /* … */;

template <class callable, class... operands>
class expr { /*...*/ };

expr<add, tridiagonal, tridiagonal>
operator+(
tridiagonal const& a, tridiagonal const& b);

expr<mul, double, tridiagonal>
operator*(double a, tridiagonal const& b);
```

Temporaries (expression templates)

```
// no calculations done here
expr<mul, double, tridiagonal> x1 = a * A;
expr<mul, double, tridiagonal> x2 = b * B;
expr<add,
    expr<mul, double, tridiagonal>,
    expr<mul, double, tridiagonal>>
        x3  = std::move(x1) + std::move(x2);

// all the calculations done in assignment
A = x3;
```

Temporaries (expression templates)

```
// no calculations done here
expr<mul, double, tridiagonal> x1 = a * A;
expr<mul, double, tridiagonal> x2 = b * B;
expr<add,
    expr<mul, double, tridiagonal>,
    expr<mul, double, tridiagonal>>
        x3  = std::move(x1) + std::move(x2);

// all the calculations done in assignment
A = x3;
```

Using C++17 to implement expression templates.

Recording Necessary Information

```
template < >
class expr {
};
```

Recording Necessary Information

```
template <class callable>
class expr {
    callable f_;
};
```

Recording Necessary Information

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
};

// for a discussion of the pronunciation of std::tuple see:
// A linear algebra library for C++23 - Guy Davidson [C++ on Sea 2019]
// https://www.youtube.com/watch?v=RzO7s-RbLwk&t=293
```

Recording Necessary Information

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
};

// Making expr a variadic template that handles all cases
// mitigates the need for CRTP
// (Curiously Recurring Template Pattern)
```

Recording Necessary Information

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    expr(callable f, operands const&... args)
     : args_(args...), f_(f) {}
};
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{ /* constructor arguments */ };
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{ /* constructor arguments */ };
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{ /* constructor arguments */ };
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      /* mul */,
      lhs, rhs};
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      [](auto const& l, auto const& r) {
          return l * r; },
      lhs, rhs};
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      [](auto const& l, auto const& r) {
          return l + r; },
      lhs, rhs};
}

// in a * A, LHS=double and RHS=tridiagonal
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      [](auto const& l, auto const& r) {
          return l * r; },
      lhs, rhs};
}
// in a * A, LHS=double and RHS=tridiagonal
// l and r are double
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      [](auto const& l, auto const& r) {
          return l * r; },
      lhs, rhs};
}
// in a * A, LHS=double and RHS=tridiagonal
// l and r are double
```

Operators Just Record Information

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs) {
    return expr{
      [](auto const& l, auto const& r) {
          return l * r; },
      lhs, rhs};
}

// LHS and RHS being unconstrained creates problems
```

Being Selective With Operands

```
template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs)
{ /* ... */ }
```

Being Selective With Operands

```
template <class LHS, class RHS>
constexpr bool is_binary_op_ok = /* … */;

template <class LHS, class RHS>
auto operator*(LHS const& lhs, RHS const& rhs)
{ /* ... */ }
```

Being Selective With Operands (SFINAE)

```
template <class LHS, class RHS>
constexpr bool is_binary_op_ok = /* … */;

template <class LHS, class RHS,
    class =
      std::enable_if_t<is_binary_op_ok<LHS, RHS>>>
auto operator*(LHS const& lhs, RHS const& rhs)
{ /* ... */ }
```

Being Selective With Operands (Concepts)

```
template <class LHS, class RHS>
constexpr bool is_binary_op_ok = /* … */;

template <class LHS, class RHS>
    requires(is_binary_op_ok<LHS, RHS>)
auto operator*(LHS const& lhs, RHS const& rhs)
{ /* ... */ }
```

Desired Calculation

```
double a, b;
tridiagonal A, B;
// …
A =
    a * A +
    b * B;
```

```
double a, b;
tridiagonal A, B;
// …
for (/* … */)
    A.v_[i] =
        a * A.v_[i] +
        b * B.v_[i];
```

Invoking the Calculation

```
class tridiagonal {
    std::vector<double> v_;
public:
    template <class src_type>
    tridiagonal& operator=(src_type const& src) {
        index const I = v_.size();
        for (index i = 0; i < I; ++i) {
            v_[i] = src[i];
        }
    }
};
```

Invoking the Calculation

```
class tridiagonal {
    std::vector<double> v_;
public:
    template <class src_type>
    tridiagonal& operator=(src_type const& src) {
        index const I = v_.size();
        for (index i = 0; i < I; ++i) {
            v_[i] = src[i];
        }
    }
};
```

Invoking the Calculation

```
class tridiagonal {
    std::vector<double> v_;
public:
    template <class src_type>
    tridiagonal& operator=(src_type const& src) {
        index const I = v_.size();
        for (index i = 0; i < I; ++i) {
            v_[i] = src[i];
        }
    }
};
```

Invoking the Calculation

```
class tridiagonal {
    std::vector<double> v_;
public:
    template <class src_type>
    tridiagonal& operator=(src_type const& src) {
        index const I = v_.size();
        for (index i = 0; i < I; ++i) {
            v_[i] = src[i];
        }
    }
};
```

Invoking the Calculation

```
class tridiagonal {
    std::vector<double> v_;
public:
    template <class src_type>
    tridiagonal& operator=(src_type const& src) {
        index const I = v_.size();
        for (index i = 0; i < I; ++i) {
            v_[i] = src[i];
        }
    }
};
```

Calculating the Results

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    auto operator[](index const i) const {
        auto const call_at_index =
            [this, i](operands const&... a) {
                return f_(subscript(a, i)...);
            };
        return std::apply(call_at_index, args_);
    }
};
```

Calculating the Results

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    auto operator[](index const i) const {
        auto const call_at_index =
            [this, i](operands const&... a) {
                return f_(subscript(a, i)...);
            };
        return std::apply(call_at_index, args_);
    }
};
```

Calculating the Results

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    auto operator[](index const i) const {
        auto const call_at_index =
            [this, i](operands const&... a) {
                return f_(subscript(a, i)...);
            };
        return std::apply(call_at_index, args_);
    }
};
```

Calculating the Results

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    auto operator[](index const i) const {
        auto const call_at_index =
            [this, i](operands const&... a) {
                return f_(subscript(a, i)...); // why not f_(a[i]...)?
            };
        return std::apply(call_at_index, args_);
    }
};
```

subscript()

```
template <class operand>
auto subscript(operand const& v, index const i) {
    if constexpr (is_array_or_expression<operand>) {
        return v[i];
    } else {
        return v;
    }
}
```

Calculating the Results

```
template <class callable, class... operands>
class expr {
    std::tuple<operands const&...> args_;
    callable f_;
public:
    auto operator[](index const i) const {
        auto const call_at_index =
            [this, i](operands const&... a) {
                return f_(subscript(a, i)...);
            };
        return std::apply(call_at_index, args_);
    }
};
```

Calculating the Results

```
template <class operand> auto
subscript(operand const& v, index const i)
{ /* calls v[i] if v is an expression */ }

template <class callable, class... operands>
class expr {
public:
    auto operator[](index const i) const {
        auto const call_at_index = [this, i](operands const&... a) {
            return f_(subscript(a, i)...);
        };
        return std::apply(call_at_index, args_);
    }
};
```

Desired Calculation

```
double a, b;
tridiagonal A, B;
// …
A =
    a * A +
    b * B;
```

```
double a, b;
tridiagonal A, B;
// …
for (/* … */)
    A.v_[i] =
        a * A.v_[i] +
        b * B.v_[i];
```

Concluding remarks.

Past, Present and Future

- Expression templates described as:
  - "beyond ugly in C++03" and
  - "still clunky in C++11."
- C++17 auto return types, class template argument deduction, lambdas and vocabulary types (std::tuple) all make expression templates easier to write.
- C++20 Concepts replace SFINAE.

Where to Learn More

- A linear algebra library for C++23 - Guy Davidson [C++ on Sea 2019]
  - https://www.youtube.com/watch?v=RzO7s-RbLwk&t=1325
- CppCon 2015: Joel Falcou PART 1 “Expression Templates - Past, Present, Future”
  - https://www.youtube.com/watch?v=IiVl5oSU5B8

Jason Turner’s C++Weekly:

- C++ Weekly - Ep 176 - Important Parts of C++11 in 12 Minutes
  - https://www.youtube.com/watch?v=D5n6xMUKU3A
- C++ Weekly - Ep 178 - The Important Parts of C++14 In 9 Minute
  - https://www.youtube.com/watch?v=mXxNvaEdNHI

THANK YOU

Optimisation & Financial Risk Analytics
Bowie Owens
Senior Software Developer

t +61 3 9545 8055
e bowie.owens@data61.csiro.au
w www.data61.csiro.au

www.data61.csiro.au

Appendices

is_array_or_expression

```
struct expression {};
template <class callable, class... operands>
class expr : public expression { /* … */ };

template <class T>
constexpr bool is_array_or_expression =
  is_array_v<T> ||
  std::is_base_of_v<expression, remove_cvref_t<T>>;

template <class A, class B>
constexpr bool is_binary_op_ok =
  is_array_or_expression<A> ||
  is_array_or_expression<B>;
```

is_array_v

```
template <class T> struct is_array {
  static constexpr bool value = false; };

template <class T>
struct is_array<std::vector<T>> {
    static constexpr bool value = true;
};

template <class T>
constexpr bool is_array_v =
    is_array<remove_cvref_t<T>>::value;
```

On Testing When Correctness is Difficult to Determine

- When refactoring we want to preserve the outcome of the software.
- If nothing else, we can:
  - run our software with given inputs,
  - record the outcome and
  - add tests that require for those inputs we get the same outcome every time.