Differentiating Object-Oriented paradigm using Clad

Petro Zarytskyi

Preserving Object Oriented Programming in Automatic Differentiation

Typical AD approaches do not preserve the object-oriented structure and abstractions in the derivative code for C++

Preserving Object Oriented Programming in Automatic Differentiation

- **Our goal** - generate derivative code that respects the original abstraction boundaries and looks like what a human developer would write

- **Why it matters** - preserving OOP structure maintains modularity, keeps derivative code readable and debuggable, and allows leveraging existing class designs and interfaces in gradient computations

- It is challenging because ...
  - The tool needs to reason about function behavior and program semantics at a high level
  - The tool must work within or alongside the compiler to access and analyze the syntactic structure
  - The approach must be easily scalable and not require case-by-case manual implementation across different OOP designs

About Clad

- **C++ source transformation -** Implemented as a compile time Clang plugin traversing the Abstract Syntax Tree (AST) of the primal function and generating the derivative code

- **Preserves original C++ syntax -** while many AD tools flatten out the compute graph to make the primal code simpler, we fully preserve the original code structure
  - Enables support for control flow expressions
  - Readable (hence easily debuggable) generated code for gradient computation
  - Compile time evaluation - templates, consteval

Note: We are only going to discuss **reverse mode AD**

Motivational Example

Example: Discrete Fourier Transform (DFT).

Hand-written derivative

Primal function

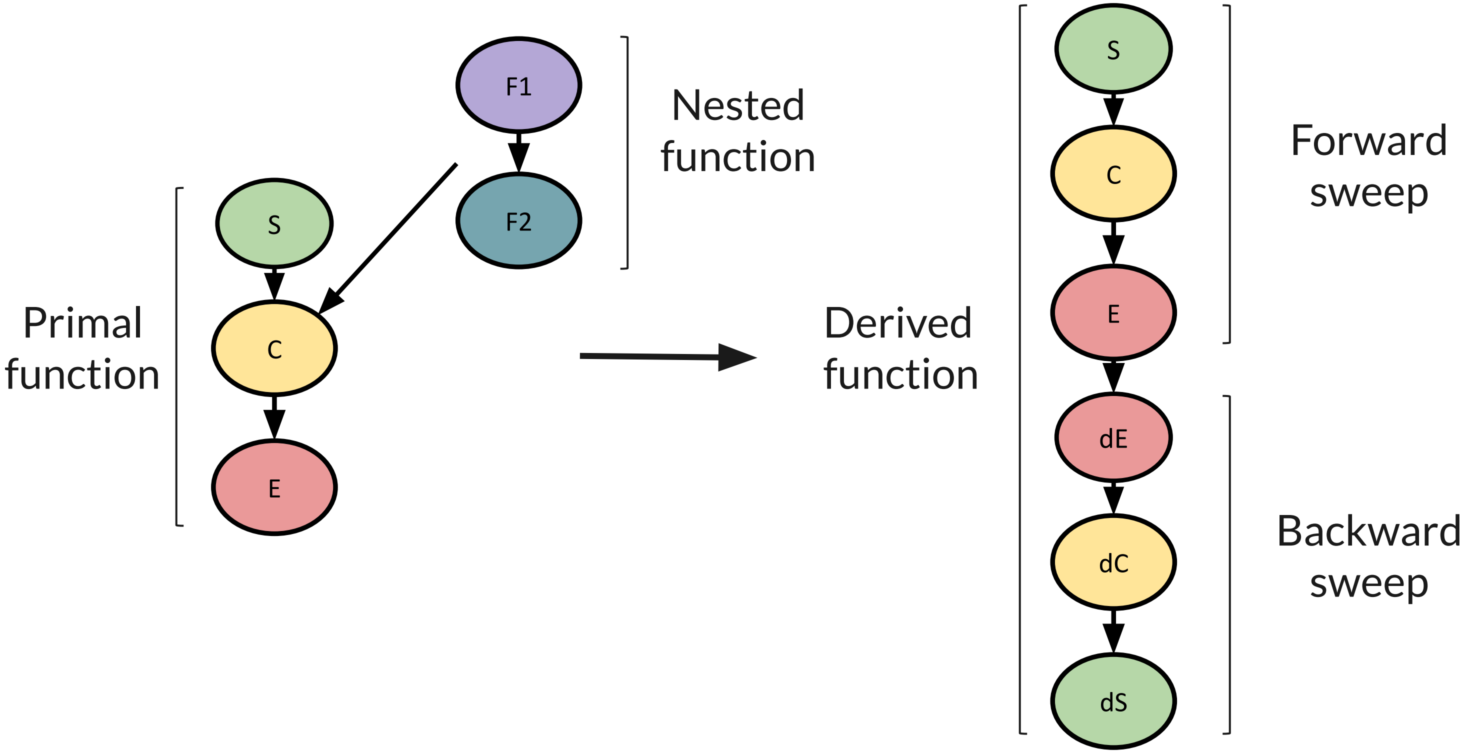
```
void dft(const std::vector<double>& signal,
         std::vector<std::complex<double>>& spectrum) {
   const std::size_t N = signal.size();
   for (std::size_t k = 0; k < N; ++k) {
      std::complex<double> sum = {0.0};
      for (std::size_t n = 0; n < N; ++n) {
         double angle = -2.0 * M_PI * k * n / N;
         std::complex<double> w(std::cos(angle),
                                std::sin(angle));
         sum += signal[n] * w;
      }
      spectrum[k] = sum;
   }
}
```

Gradient

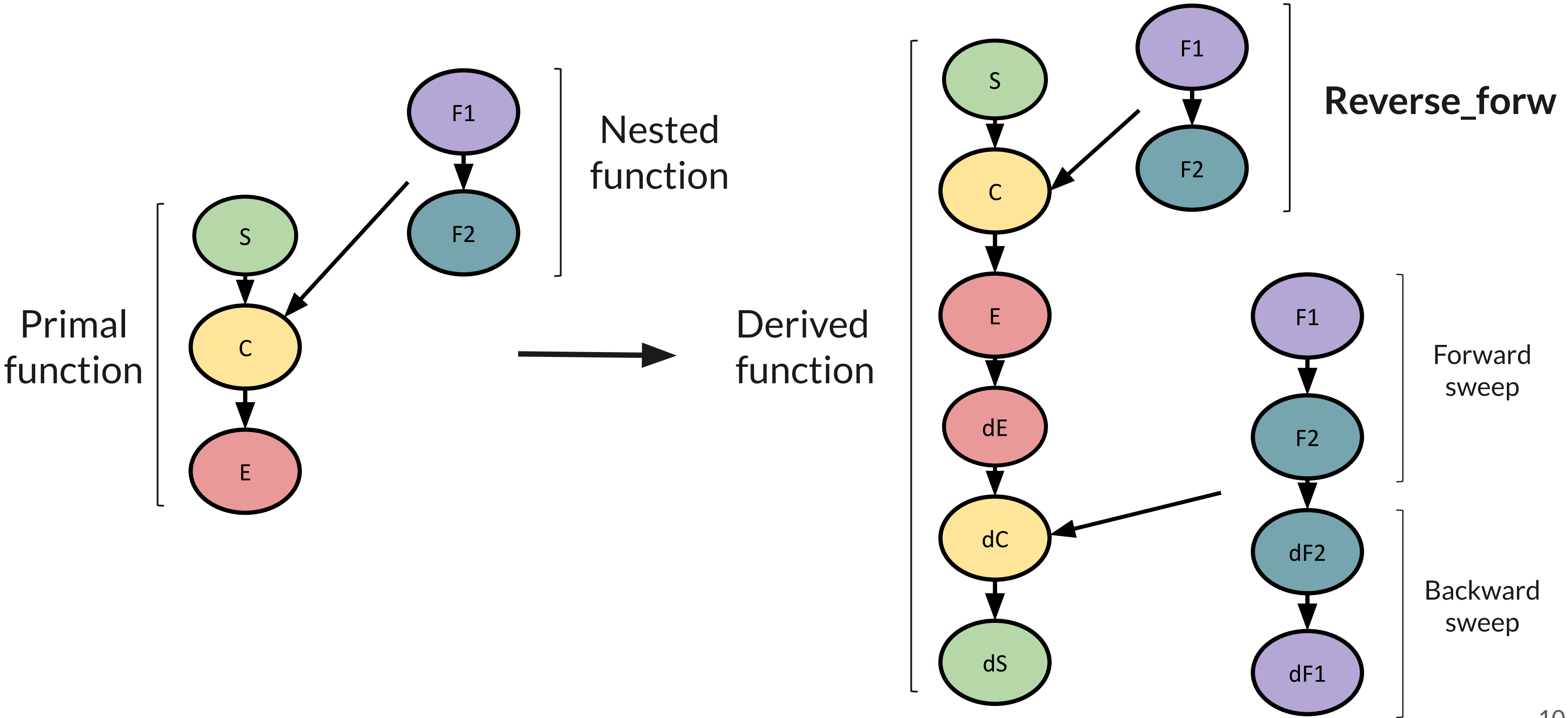
```
void dft_grad(const std::vector<double>& signal,
              std::vector<std::complex<double>>& spectrum,
              std::vector<double>* d_signal,
              std::vector<std::complex<double>>* d_spectrum)
{
   const std::size_t N = signal.size();
   for (std::size_t k = 0; k < N; ++k) {
      std::complex<double> d_sum = (*d_spectrum)[k];
      for (std::size_t n = 0; n < N; ++n) {
         double angle = -2.0 * M_PI * k * n / N;
         std::complex<double> w(std::cos(angle),
                                std::sin(angle));
         (*d_signal)[n] += std::real(d_sum * std::conj(w));
      }
   }
}
```

Compute Graph of Derivative Code

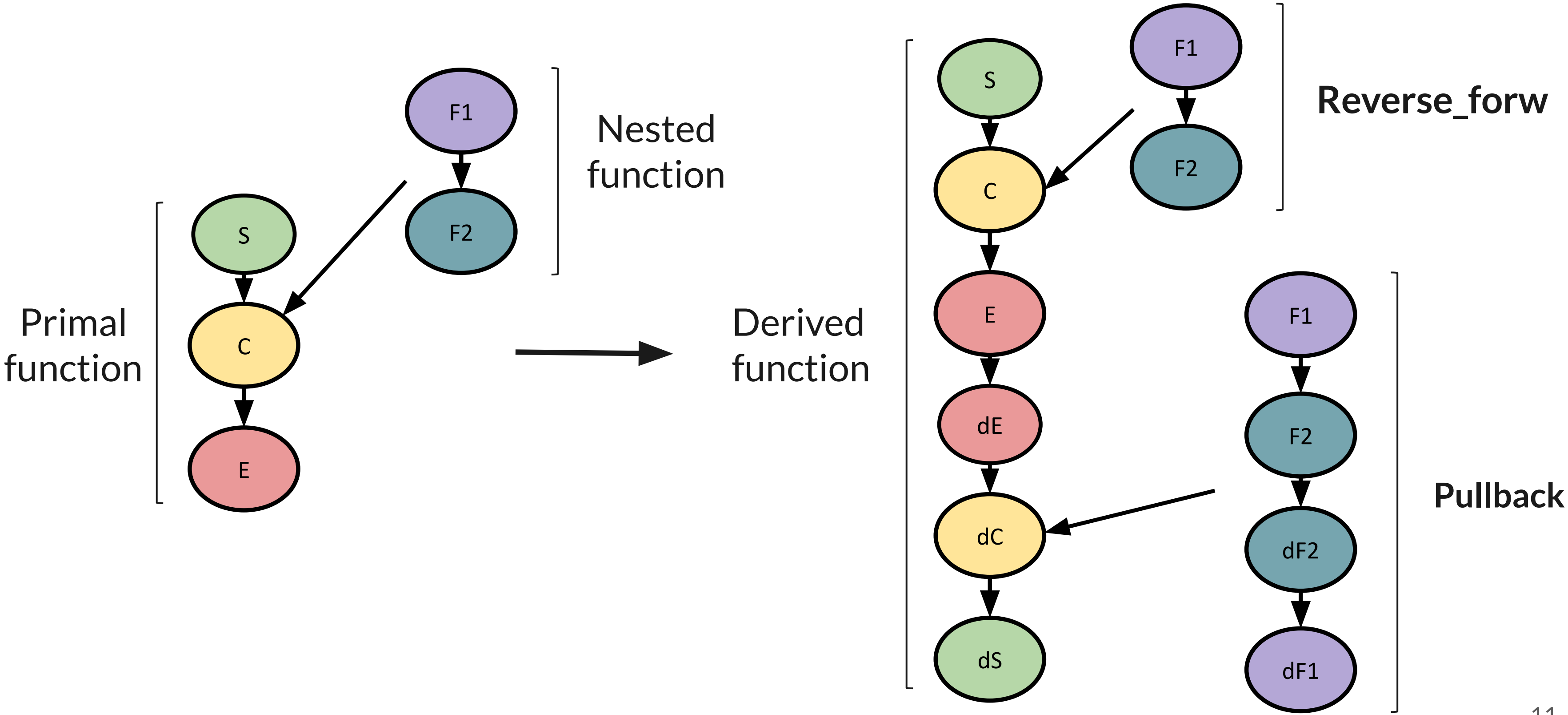
Compute Graph of Derivative Code



Compute Graph of Derivative Code

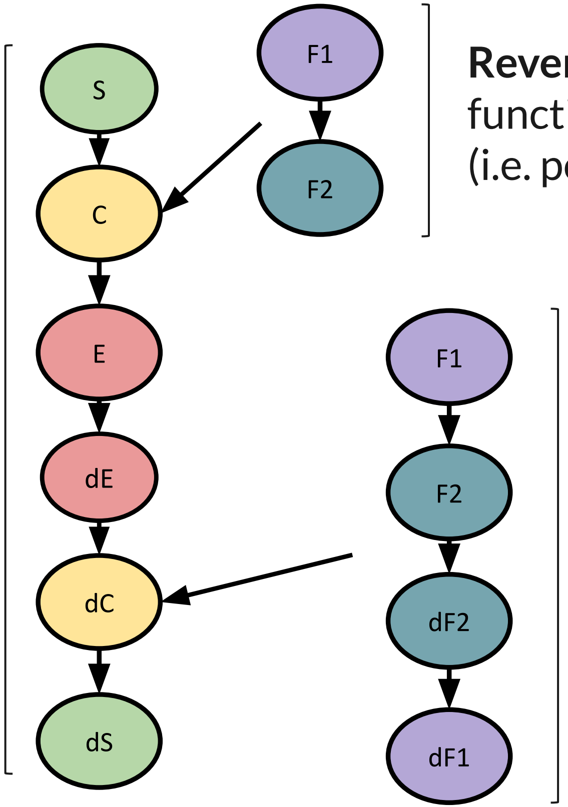


Compute Graph of Derivative Code



Compute Graph of Derivative Code

Derived function



Reverse_forw - forward sweep of the function. Handles memory operations (i.e. pointer/reference adjoints)

Pullback - reverse sweep of the function. Handles numerical operations

Differentiating the example automatically

Example: Discrete Fourier Transform (DFT).

Automatically generated derivatives

Primal function

```cpp
void dft(const std::vector<double>& signal,
         std::vector<std::complex<double>>& spectrum) {
    const std::size_t N = signal.size();
    for (std::size_t k = 0; k < N; ++k) {
        std::complex<double> sum = {0.0};
        for (std::size_t n = 0; n < N; ++n) {
            double angle = -2.0 * M_PI * k * n / N;
            std::complex<double> w(std::cos(angle),
                                   std::sin(angle));
            sum += signal[n] * w;
        }
        spectrum[k] = sum;
    }
}
```

Gradient *18 functions 244 lines*

```cpp
static constexpr void constructor_pullback(double __re, double __im, std::complex<double> *_d_this, double *_d___re, double *_d___im) {...}
void size_pullback(size_type _d_y, std::vector<double> *_d_this) const noexcept {...}
constexpr void real_pullback(double _d_y, std::complex<double> *_d_this) const {...}
constexpr void imag_pullback(double _d_y, std::complex<double> *_d_this) const {...}
void operator_subscript_pullback(size_type __n, value_type _d_y, std::vector<std::complex<double> > *_d_this, size_type *_d___n) noexcept
{...}
void operator_subscript_pullback(size_type __n, value_type _d_y, std::vector<std::complex<double> > *_d_this, size_type *_d___n) noexcept
{...}
clad::ValueAndAdjoint<reference, reference> operator_subscript_reverse_forw(size_type __n, std::vector<std::complex<double> > *_d_this,
size_type _d___n) noexcept {...}
clad::ValueAndAdjoint<reference, reference> operator_subscript_reverse_forw(size_type __n, std::vector<std::complex<double> > *_d_this,
size_type _d___n) noexcept {...}
clad::ValueAndAdjoint<complex<double> &, complex<double> &> operator_plus_equal_reverse_forw(const complex<double> &__c,
std::complex<double> *_d_this, const complex<double> &_d___c, clad::restore_tracker&_tracker0) {...}
void operator_plus_equal_pullback(const complex<double> &__c, std::complex<double> *_d_this, complex<double> *_d___c) {...}
static inline constexpr void constructor_pullback(const complex<double> &arg0, std::complex<double> *_d_this, complex<double> *_d_arg0)
noexcept {...}
clad::ValueAndAdjoint<complex<double> &, complex<double> &> operator_star_equal_reverse_forw(double __re, std::complex<double> *_d_this,
double _d___re, clad::restore_tracker&_tracker0) {...}
operator_star_equal_pullback(double __re, std::complex<double> *_d_this, double *_d___re) {...}
static inline constexpr void constructor_pullback(complex<double> &&arg0, std::complex<double> *_d_this, complex<double> *_d_arg0) noexcept
{...}
inline void operator_star_pullback(const double &__x, const complex<double> &__y, complex<double> _d_y, double *_d___x, complex<double>
*_d___y) {...}
inline constexpr clad::ValueAndAdjoint<complex<double> &, complex<double> &> operator_equal_reverse_forw(const complex<double> &arg0,
std::complex<double> *_d_this, const complex<double> &_d_arg0, clad::restore_tracker&_tracker0) noexcept {...}
inline constexpr void operator_equal_pullback(const complex<double> &arg0, std::complex<double> *_d_this, complex<double> *_d_arg0)
noexcept {...}

void dft_grad(const std::vector<double> &signal, std::vector<std::complex<double> > &spectrum, std::vector<double> *_d_signal,
std::vector<std::complex<double> > *_d_spectrum) {...}
```

So why did we get so many derivatives?

```
void dft(const std::vector<double>& signal,
         std::vector<std::complex<double>>& spectrum) {
    const std::size_t N = signal.size();
    for (std::size_t k = 0; k < N; ++k) {
        std::complex<double> sum = {0.0};
        for (std::size_t n = 0; n < N; ++n) {
            double angle = -2.0 * M_PI * k * n / N;
            std::complex<double> w(std::cos(angle),
                std::sin(angle));
            sum += signal[n] * w;
        }
        spectrum[k] = sum;
    }
}
```

All of these are hidden function calls that require reverse_forw and pullback

OOP-motivated optimizations. Expressing Semantics

How can we avoid generating these derivatives?

Consider a simple example

```
std::vector<double> vec;
vec[i] = x;
```

→

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
auto _t0 = operator_subscript_reverse_forw(vec, i, _d_vec, _d_i);
_t0.value = x;

// reverse pass
operator_subscript_pullback(vec, i, _d_sum, &_d_vec, &_d_i);
```

```
double& std::vector<double>::operator[](size_t i)
{...}
```

→

```
clad::ValueAndAdjoint<double&, double&>
operator_subscript_reverse_forw(...) {...}
```

→

```
void operator_subscript_pullback(...) {...}
```

Step 1: Remove pullbacks of access-only functions

Such operations can be expressed with reverse_forw

```
std::vector<double> vec;
vec[i] = x;
```

→

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
auto _t0 = operator_subscript_reverse_forw(vec, i, _d_vec, _d_i);
_t0.value = x;

// reverse pass
~~operator_subscript_pullback(vec, i, _d_sum, &_d_vec, &_d_i);~~  _d_x += _t0.adjoint;
```

```
double& std::vector<double>::operator[](size_t i)
{...}
```

→

```
clad::ValueAndAdjoint<double&, double&>
operator_subscript_reverse_forw(...) {...}
```

→

```
~~void operator_subscript_pullback(...) {...}~~
```

Step 1: Remove pullbacks of access-only functions

Such operations can be expressed with reverse_forw

```
std::vector<double> vec;
vec[i] = x;
```

→

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
auto _t0 = operator_subscript_reverse_forw(vec, i, _d_vec, _d_i);
_t0.value = x;

// reverse pass
_d_x += _t0.adjoint;
```

```
double& std::vector<double>::operator[](size_t i)
{...}
```

→

```
clad::ValueAndAdjoint<double&, double&>
operator_subscript_reverse_forw(...) {...}
```

This is done automatically now!

Step 2: Elide the reverse_forw

```
std::vector<double> vec;
vec[i] = x;
```

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
auto _t0 = operator_subscript_reverse_forw(vec, i, _d_vec, _d_i);
_t0.value = x;

// reverse pass
_d_x += _t0.adjoint;
```

```
double& std::vector<double>::operator[](size_t i)
{...}
```

```
clad::ValueAndAdjoint<double&, double&>
operator_subscript_reverse_forw(...) {...}
```

Notice: this is just ***{vec[i], _d_vec[i]}***

Step 2: Elide the reverse_forw

```
std::vector<double> vec;
vec[i] = x;
```

→

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
auto _t0 = operator_subscript_reverse_forw(vec, i, _d_vec, _d_i);
_t0.value = x; vec[i] = x;

// reverse pass
_d_x += _t0.adjoint; d_x += _d_vec[i];
```

```
double& std::vector<double>::operator[](size_t i)
{...}
```

→

```
clad::ValueAndAdjoint<double&, double&>
operator_subscript_reverse_forw(...) {...}
```

Notice: this is just ***{vec[i], _d_vec[i]}***

Step 2: Elide the reverse_forw

```
std::vector<double> vec;
vec[i] = x;
```

→

```
std::vector<double> vec;
std::vector<double> _d_vec;
// forward pass
vec[i] = x;


// reverse pass
d_x += _d_vec[i];
```

For now can only be **requested manually** with the **elidable_reverse_forw attribute**

```
clad::ValueAndAdjoint<T&, T&>
operator_subscript_reverse_forw(std::vector<T>* vec, ...) elidable_reverse_forw;
```

How do these changes impact the previous example?

- Hand-written Gradient: 1 function 15 lines
- Automatic gradient (no optimization): 18 functions 244 lines
- **Automatic gradient (optimized): 12 functions 202 lines**

Summary

- **Promising results with semantic awareness** - our approach shows preserving class structures and semantics can lead to significant derivative code simplification

- **A path forward for automated optimization -** while some optimization requires manual intervention, we've demonstrated the feasibility and effectiveness of this approach, paving the way for the upcoming automation

- **Future work:**
  - **Broadening container coverage -** extending our analysis to encompass standard accessor functions (operator[], front(), back(), operator*) across common containers (std::vector, std::list) and smart pointers (std::unique_ptr, std::shared_ptr)
  - **Expanding functional scope -** moving beyond pure functions to handle side-effect operations (like std::vector::push_back) and constructors

Thank you!