IMPROVING AUTOMATIC DIFFERENTIATION OF OBJECT-ORIENTED PARADIGMS USING CLAD

Daemon
WHAT WE ACHIEVED

• Enhanced the clad object-oriented differentiable model by incorporating non-differentiable attributes.
• Introduced support for reference return types in clad's reverse mode.
• Upon facilitating the reference return type, we also activated operator overloading in both forward and reverse modes.
• Enabled user-defined derivative functions for operator overloads.
WHAT WE ACHIEVED

• https://github.com/vgvassilev/clad/pull/568
• https://github.com/vgvassilev/clad/pull/605
• https://github.com/vgvassilev/clad/pull/601 (complete, waiting to be merged)
• https://github.com/vgvassilev/clad/pull/619 (complete, will be rebased on PR601)
• NON-DIFFERENTIABLE ATTRIBUTES.

```cpp
non_differentiable double product(double value) {
    return x * y * value;
}
double mem_fn(double value) {
    return product(value) * value;
}
```

• non_differentiable is an attribute that marks specific fields or methods in a class, indicating they should not be differentiated.

• Here, the product method in the SimpleFunctions class has been tagged with this attribute, signifying that any differentiation tools or routines should bypass or ignore this method.
• NON-DIFFERENTIABLE ATTRIBUTES.

```cpp
non_differentiable double product(double value) {
    return x * y * value;
}
double mem_fn(double value) {
    return product(value) * value;
}
```

```cpp
clad::ValueAndPushforward<double, double> mem_fn_pushforward(double value,
    | SimpleFunctions * _d_this, double _d_value) {
    double _t0 = this->product(value);
    return {_t0 * value, 0 * value + _t0 * _d_value};
}
```
• NON-DIFFERENTIABLE ATTRIBUTES.

```cpp
class non_differentiable SimpleFunctions2 {
public:
    SimpleFunctions2() noexcept : x(0), y(0) {}  
    SimpleFunctions2(double p_x, double p_y) noexcept : x(p_x), y(p_y) {}  
    double x;
    double y;
    double mem_fn(double i, double j) { return (x + y) * i + i * j * j; }
    SimpleFunctions2 operator+(const SimpleFunctions2& other) const {
        return SimpleFunctions2(x + other.x, y + other.y);
    }
};
```

• When applied to a class, it suggests that differentiation tools should bypass or ignore all of its fields and member functions.
• OPERATOR OVERLOADS

```cpp
SimpleFunctions& operator+=(double value) {
    x += value;
    return *this;
}
```

```cpp
double fn2(SimpleFunctions& v, double value) {
    v += value;
    return v.x;
}
```

```cpp
auto fn2_grad = clad::gradient(fn2);
```

• The above example demonstrates the differentiation of operator overloads using clad.

• A crucial enhancement added is the support for operators with reference return types, such as the operator+= in the SimpleFunctions class.
A crucial enhancement added is the support for operators with reference return types, such as the operator+= in the SimpleFunctions class.

We introduce a “_forw” function for reference return type.
• REFERENCE RETURN TYPE

```cpp
// derivative declarations
double _d_a = 0;
double& _d_a_ref = _d_a;

// forward pass
double& a_ref = a;
```

• In the above example, we can easily point _d_a_ref to _d_a because the derivative of a is known at compile time. This is not always the case, for example, consider the following code.

```cpp
dooble& someFn(double& i, double&j, double& k) { ... }
dooble fn(double i, double j, double k) {
    double& ref = someFn(i, j, k);
}
• REFERENCE RETURN TYPE

```cpp
double& someFn(double& i, double& j, double& k) { ... }

double fn(double i, double j, double k) {
    double& ref = someFn(i, j, k);
}
```

• We cannot determine which variable ref is referencing at compile time. Thus, we also cannot determine which derivative should _d_ref refer to.

• That’s why we need “_forw” function.
• **REFERENCE RETURN TYPE**

```cpp
double& someFn(double& i, double& j) {
    double& k = i;
    double& l = j;
    if (...) {
        return k;
    } else {
        return l;
    }
}
```

// derivative declarations
double* _d_ref = nullptr;

// forward pass
double t0 = i;
double t1 = j;
clad::ValueAndAdjoint<double&, double&> t = someFn_forw(i, j, &_d_i, &_d_j);
_d_ref = &t.adjoint;
double& ref = t.value;

// reverse pass
someFn_pullback(t0, t1, /*pullback=*/double(), &_d_i, &_d_j);
```
• The corresponding `someFn_forw` will be:

```cpp
double& someFn(double& i, double& j) {
    double& k = i;
    double& l = j;
    if (...)  
        return k;
    else
        return l;
}
```

```cpp
clad::ValueAndAdjoint<double, double>
someFn_forw(double& i, double& j, clad::array_ref<double> _d_i,
             double* _d_k = nullptr;
             double* _d_l = nullptr;

    // forward pass
    _d_k = &* _d_i;
    double& k = i;
    _d_l = &* _d_j;
    double& l = j;

    if (...)  
        return {k, * _d_k};
    else
        return {l, * _d_l};
```
• CUSTOM DERIVATIVES FOR SPECIAL MEMBER FUNCTIONS

• The code showcases user-defined derivatives for operator overloads, allowing for custom differentiation behavior.

• By employing the clad::custom_derivatives namespace, users can specify custom derivatives for operators like operator+=, tailoring differentiation to specific class implementations.
MISSING SUPPORT FOR CPP FEATURE

• Support try-catch blocks to enable some std namespace functions differentiation.

• Support switch statements in the reverse mode.

• Support special member functions like constructors in both the forward and the reverse mode.

• Support custom derivatives for special member functions.
THANK YOU!