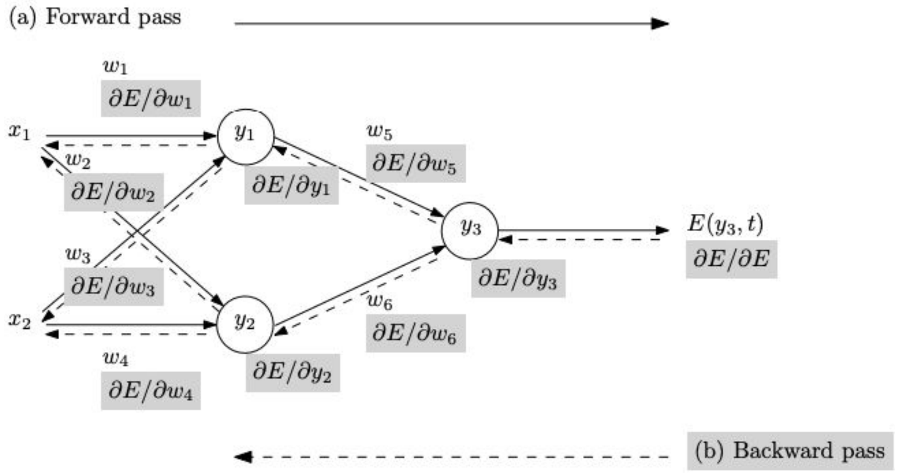
Improve automatic differentiation of object-oriented paradigms using Clad

Petro Zarytskyi
Google Summer of Code
Julius-Maximilians-Universität, Germany
Mentors: Vassil Vassilev, David Lange

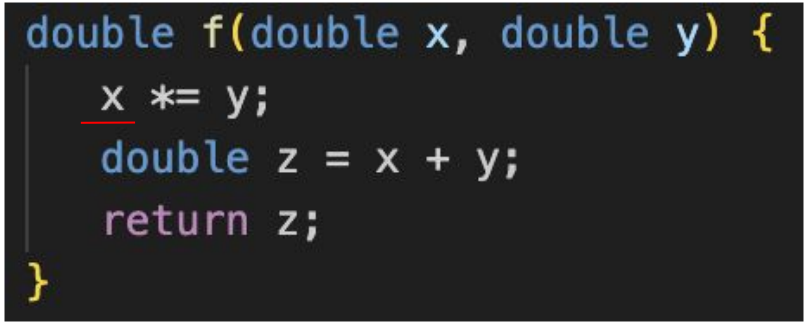
Introduction: Automatic Differentiation

Automatic differentiation is a method of differentiation of functions expressed as procedures. It involves breaking up the function into simple operations and applying chain rule to each one of them. This can be done both ways: from the input to the output (forward mode) and vice versa (reverse mode). This project focuses on the second approach which is more efficient for computing gradients. In reverse mode, we need two passes: a forward pass to store the intermediate values of all the variables and a backward pass to compute derivatives.


(a) Forward pass
w_1
$\partial E/\partial w_1$
x_1
w_2
$\partial E/\partial w_2$
y_1
$\partial E/\partial y_1$
w_5
$\partial E/\partial w_5$
y_3
$\partial E/\partial y_3$
$E(y_3, t)$
$\partial E/\partial E$
w_3
$\partial E/\partial w_3$
x_2
y_2
$\partial E/\partial y_2$
w_6
$\partial E/\partial w_6$
w_4
$\partial E/\partial w_4$
(b) Backward pass

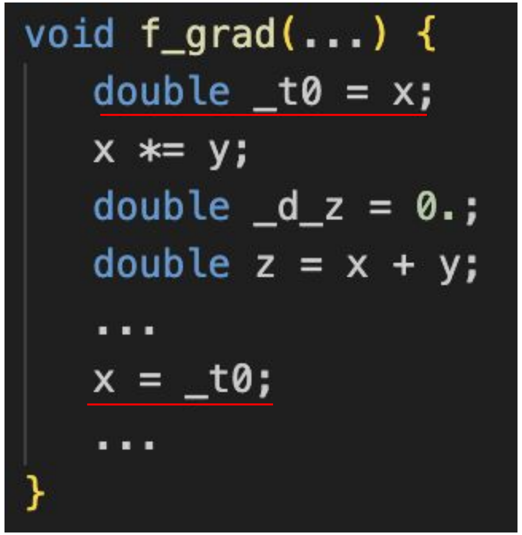
Examples: what works

Original code



```
double f(double x, double y) {
   x *= y;
   double z = x + y;
   return z;
}
```

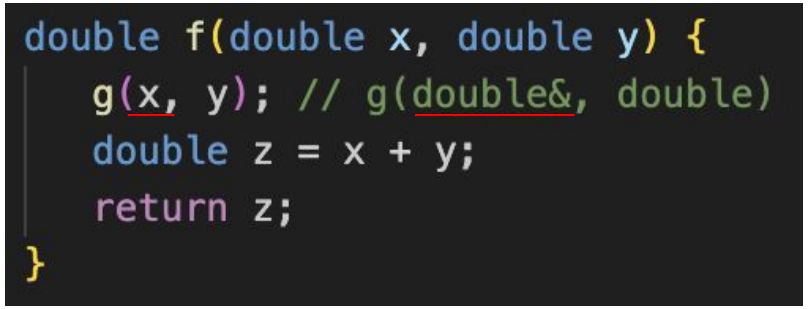
Code differentiated by Clad



```
void f_grad(...) {
   double _t0 = x;
   x *= y;
   double _d_z = 0.;
   double z = x + y;
   ...
   x = _t0;
   ...
}
```

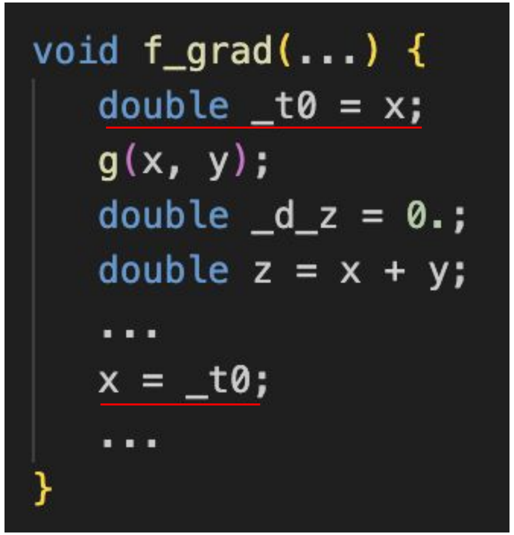
Examples: what works

Original code



```
double f(double x, double y) {
    g(x, y); // g(double&, double)
    double z = x + y;
    return z;
}
```

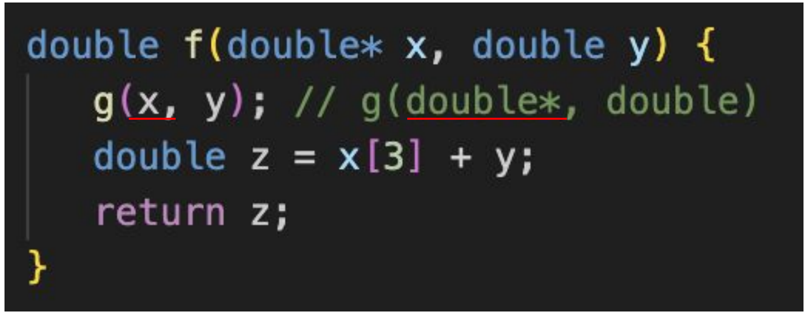
Code differentiated by Clad



```
void f_grad(...) {
    double _t0 = x;
    g(x, y);
    double _d_z = 0.;
    double z = x + y;
    ...
    x = _t0;
    ...
}
```

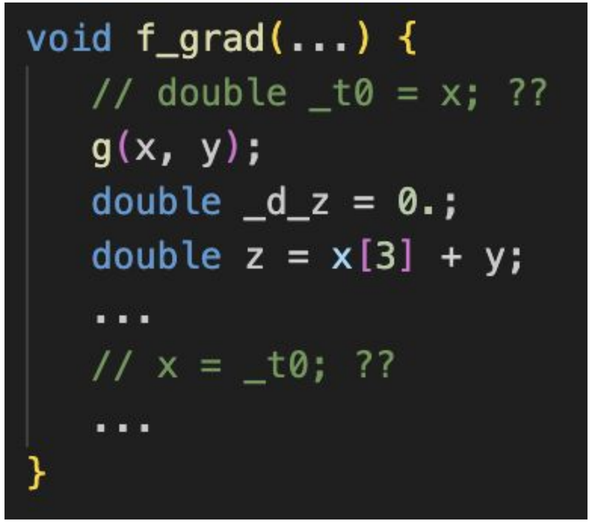
Examples: what doesn't work

Original code



```
double f(double* x, double y) {
    g(x, y); // g(double*, double)
    double z = x[3] + y;
    return z;
}
```

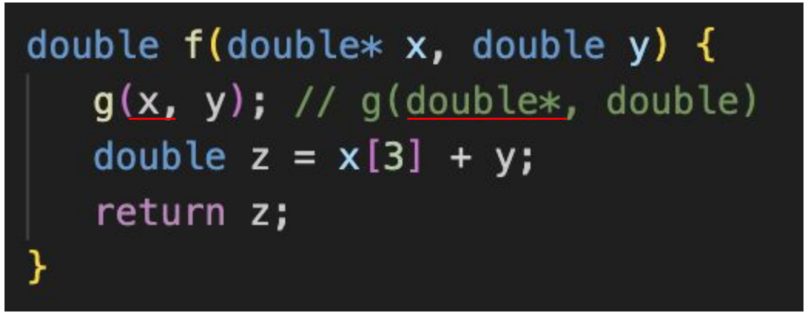
Code differentiated by Clad



```
void f_grad(...) {
    // double _t0 = x; ??
    g(x, y);
    double _d_z = 0.;
    double z = x[3] + y;
    ...
    // x = _t0; ??
    ...
}
```

Examples: what doesn't work

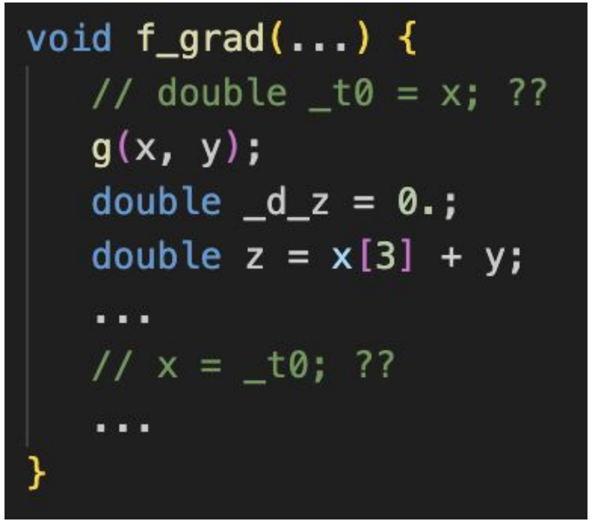
Original code



```
double f(double* x, double y) {
    g(x, y); // g(double*, double)
    double z = x[3] + y;
    return z;
}
```

C-arrays are not copyable!

Code differentiated by Clad



```
void f_grad(...) {
    // double _t0 = x; ??
    g(x, y);
    double _d_z = 0.;
    double z = x[3] + y;
    ...
    // x = _t0; ??
    ...
}
```

What doesn’t work

The same goes with all non-copyable types:

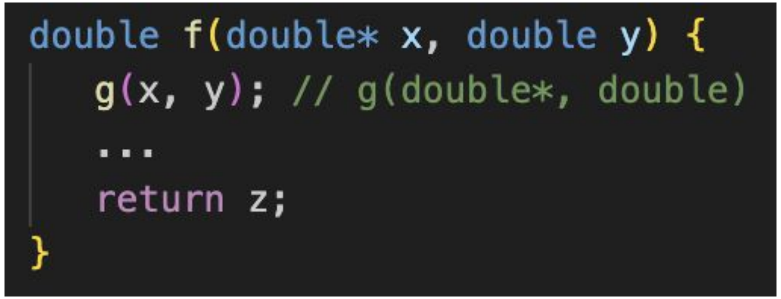
- std::initializer_list
- std::unique_ptr, std::shared_ptr, etc.
- Other STL and user-defined types.

Note: Methods of such classes follow the same logic as plain functions.

Approach 1: general solution

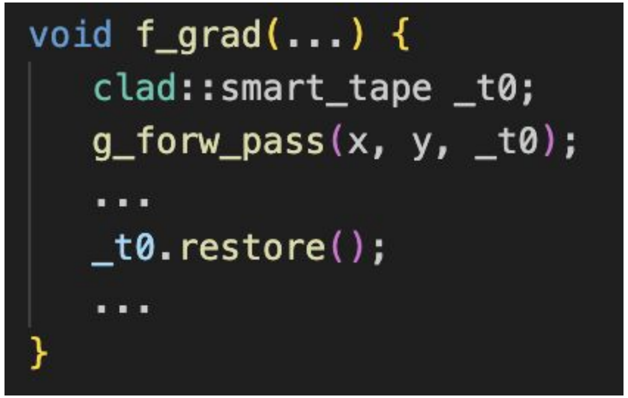
We introduce a new type of non-copyable types that can automatically store and restore multiple objects at the same time

Original code



```
double f(double* x, double y) {
    g(x, y); // g(double*, double)
    ...
    return z;
}
```

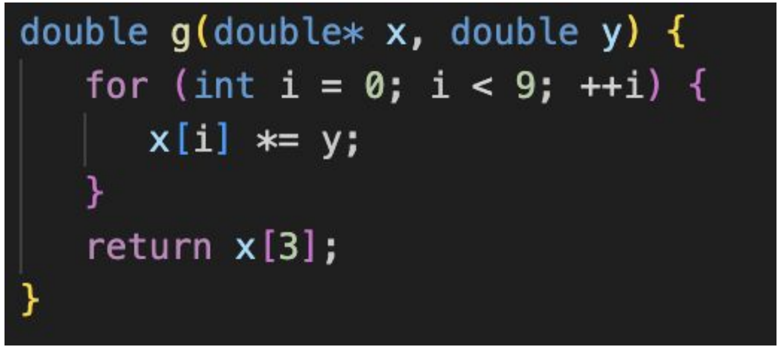
Code differentiated by Clad



```
void f_grad(...) {
    clad::smart_tape _t0;
    g_forw_pass(x, y, _t0);
    ...
    _t0.restore();
    ...
}
```

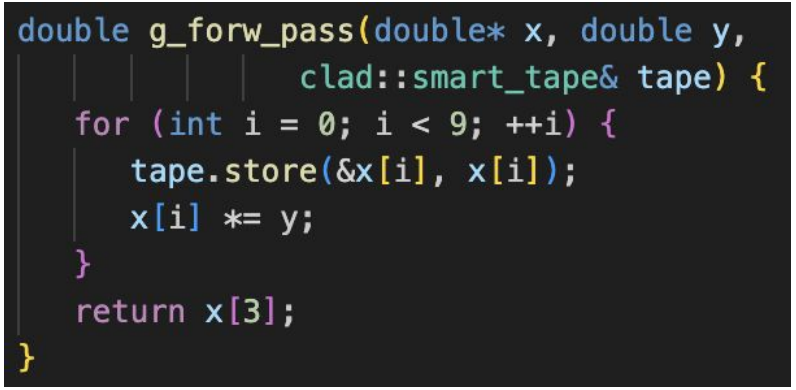
Approach 1: general solution

Original code



```
double g(double* x, double y) {
    for (int i = 0; i < 9; ++i) {
        x[i] *= y;
    }
    return x[3];
}
```

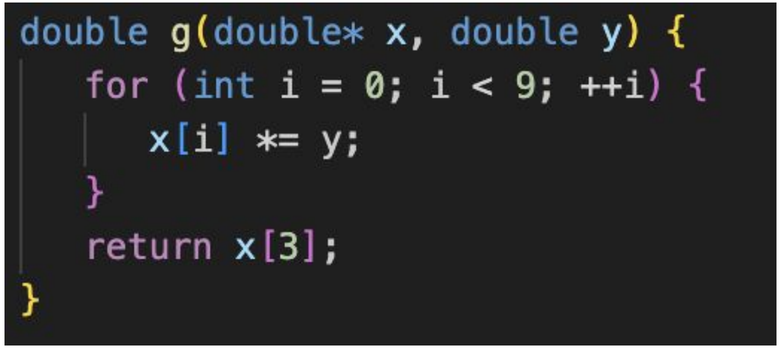
Code differentiated by Clad



```
double g_forw_pass(double* x, double y,
                   clad::smart_tape& tape) {
    for (int i = 0; i < 9; ++i) {
        tape.store(&x[i], x[i]);
        x[i] *= y;
    }
    return x[3];
}
```

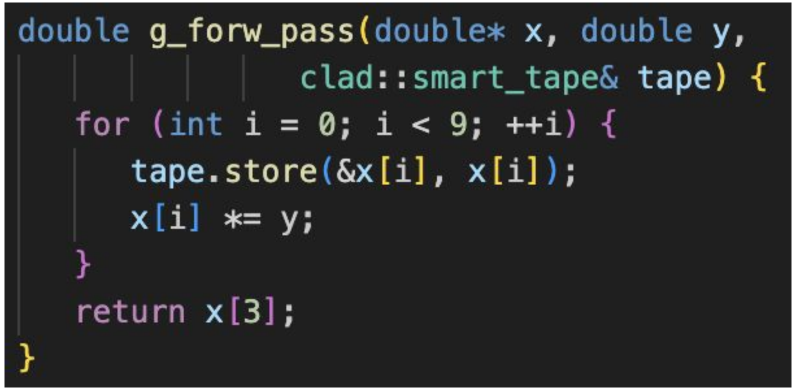
Approach 1: general solution

Original code



```
double g(double* x, double y) {
    for (int i = 0; i < 9; ++i) {
        x[i] *= y;
    }
    return x[3];
}
```

Code differentiated by Clad



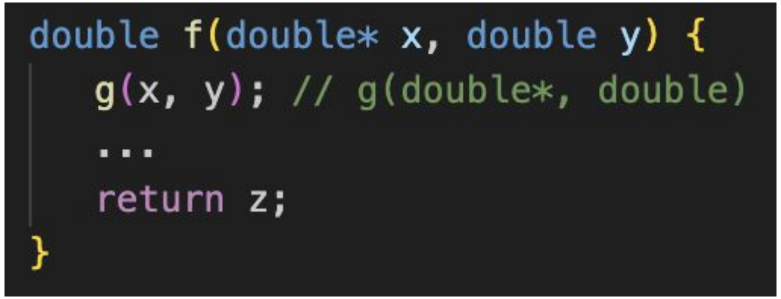
```
double g_forw_pass(double* x, double y,
                   clad::smart_tape& tape) {
    for (int i = 0; i < 9; ++i) {
        tape.store(&x[i], x[i]);
        x[i] *= y;
    }
    return x[3];
}
```

Not very efficient / readable but simple and works

Approach 2: nice partial solution

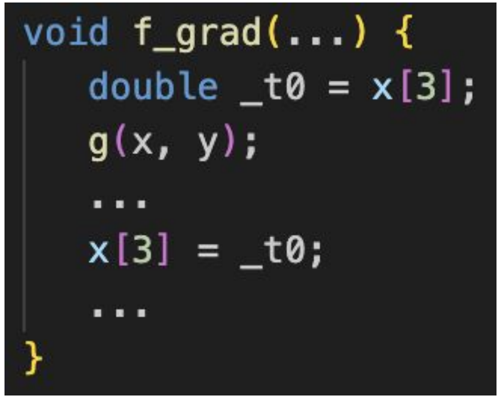
Let's say we have analysed g and we know that it only affects the third element of x

Original code



```
double f(double* x, double y) {
    g(x, y); // g(double*, double)
    ...
    return z;
}
```

Code differentiated by Clad



```
void f_grad(...) {
    double _t0 = x[3];
    g(x, y);
    ...
    x[3] = _t0;
    ...
}
```

Approach 2: nice partial solution

Good news:

This type of analysis already exists in Clad and it's called TBR (To-Be-Recorded)

Bad news:

It's quite unreliable, especially with arrays/structures and nested functions

Summary

Approach 1:

- General
- Straightforward implementation
- A little ugly
- Can still benefit from TBR to avoid unnecessary stores

Approach 2:

- Partial
- Sophisticated implementation
- More readable/efficient code
- Relies on improvements to TBR

Ideally, we should implement both and have the first one as fallback behavior.
The exact order of implementation is up to a discussion.

Successfully implementing this system will not only enable differentiation of non-copyable structures but also make storing all structures more efficient.

Thank you!