



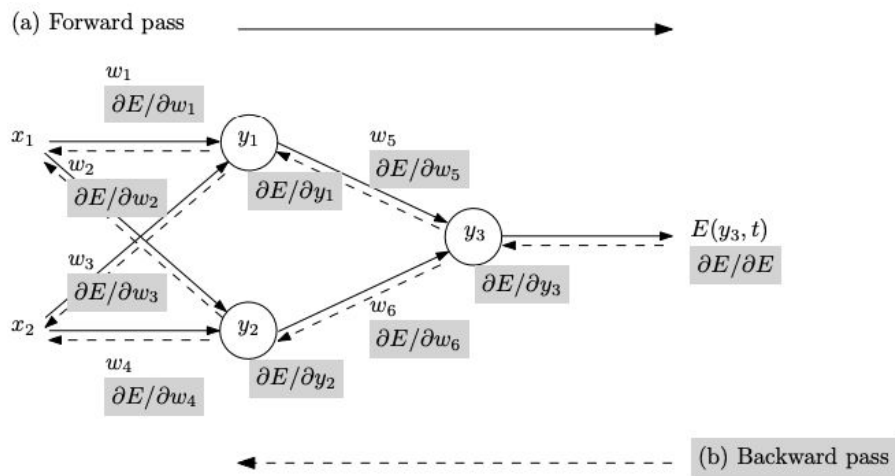
# Optimizing reverse-mode automatic differentiation with advanced activity analysis

Petro Zarytskyi  
IRIS-HEP Fellow

Taras Shevchenko National University of Kyiv, Ukraine  
Mentors: Vassil Vassilev, David Lange

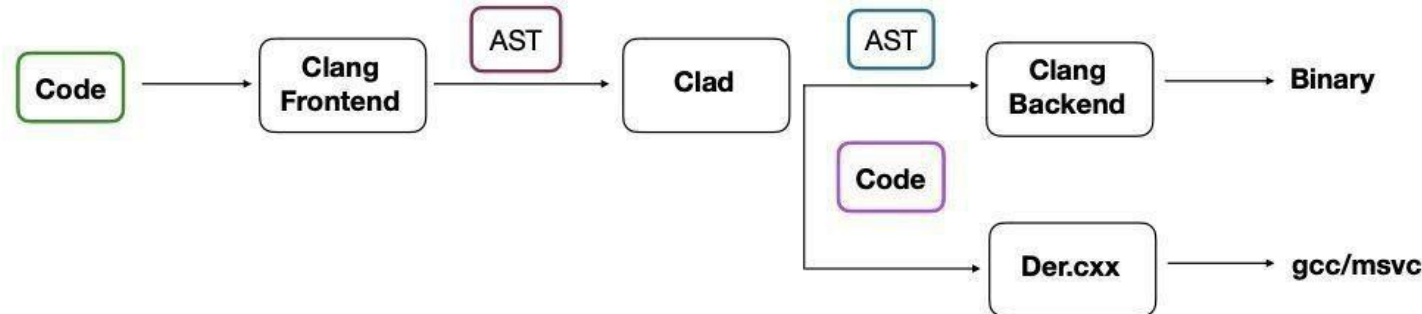
# Introduction: Automatic Differentiation

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.



# Introduction: Clad

Clad is an automatic differentiation Clang plugin for C++. It automatically generates code that computes derivatives of functions given by the user.



```
double f(double x) {
    return x * x;
}
```

```
FunctionDecl f 'double (double)'
  -ParamVarDecl x 'double'
  -CompoundStmt
  -ReturnStmt
    -BinaryOperator 'double' '*'
      -ImplicitCastExpr 'double' <LValueToRValue>
        -DeclRefExpr 'double' lvalue ParamVar 'x' 'double'
      -ImplicitCastExpr 'double' <LValueToRValue>
        -DeclRefExpr 'double' lvalue ParamVar 'x' 'double'
```

```
FunctionDecl 0x7f7f801d0f00 <<invalid sloc> <<invalid sloc> f_darg0 'double (double)'
  -ParamVarDecl 0x7f7f801d0d80 <<invalid sloc> <<invalid sloc> 'double' 'd_x' 'double'
  -CompoundStmt 0x7f7f801d0c30 <<invalid sloc>
    -DeclRefExpr 0x7f7f801d0b10 <<invalid sloc>
      -VarDecl 0x7f7f801d0a10 <<invalid sloc> <<invalid sloc> used '_d_x' 'double' 'const'
      -ImplicitCastExpr 0x7f7f801d0910 <<invalid sloc> 'double' <<integralToInteger>
      -UnaryOperator 0x7f7f801d0810 <<invalid sloc> 'int' '1'
      -ReturnStmt 0x7f7f801d0710 <<invalid sloc>
        -BinaryOperator 0x7f7f801d0610 <<invalid sloc> 'double' '*'
          -BinaryOperator 0x7f7f801d0510 <<invalid sloc> 'double' '+', 2, 0x7f7f801d0410 <<invalid sloc>
            -ImplicitCastExpr 0x7f7f801d0310 <<invalid sloc> 'double' <<LValueToRValue>
              -DeclRefExpr 0x7f7f801d0210 <<invalid sloc> 'double' 'd_x' 'double'
            -ImplicitCastExpr 0x7f7f801d0110 <<invalid sloc> 'double' <<LValueToRValue>
              -DeclRefExpr 0x7f7f801d0010 <<invalid sloc> 'double' 'x' 'double'
          -DeclRefExpr 0x7f7f801c0f10 <<invalid sloc> 'double' 'd_x' 'double'
        -DeclRefExpr 0x7f7f801c0e10 <<invalid sloc> 'double' 'd_x' 'double'
      -DeclRefExpr 0x7f7f801c0d10 <<invalid sloc> 'double' 'd_x' 'double'
    -DeclRefExpr 0x7f7f801c0c10 <<invalid sloc> 'double' 'd_x' 'double'
  -DeclRefExpr 0x7f7f801c0b10 <<invalid sloc> 'double' 'd_x' 'double'
```

```
double f_darg0(double x) {
    double _d_x = 1;
    return _d_x * x + x * _d_x;
}
```



# Activity Analysis

This includes removing:

- The derivatives of variables that don't depend on the input values in a differentiable way
- The derivatives of variables that don't influence the output values in a differentiable way
- Variables that are not used to compute the derivatives of the output values

As well as possibly:

- Removing computations that are not needed to evaluate the final derivatives
- Simplifying operations with implicit zeros



# TBR (To-Be-Recorded) Analysis

Reverse-mode automatic differentiation requires storing intermediate values of variables that have impact on derivatives to restore those in the backward pass. However, we don't actually have to store all of them.

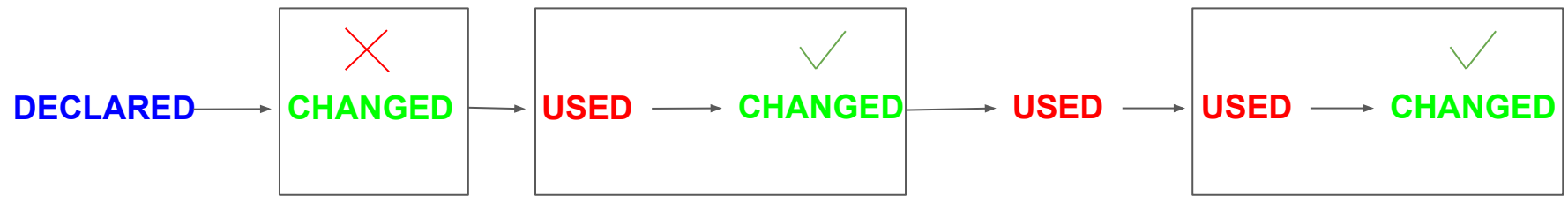
## History of usage of a variable x

DECLARED → CHANGED → USED → CHANGED → USED → USED → CHANGED

# TBR (To-Be-Recorded) Analysis

Reverse-mode automatic differentiation requires storing intermediate values of variables that have impact on derivatives to restore those in the backward pass. However, we don't actually have to store all of them.

## History of usage of a variable x



# Examples

## Original code

```
double f(double x, double y) {  
    double k = 2.5;  
    x += k;  
    return x;  
}
```

k doesn't depend on x and y in a differentiable way!

## Code differentiated by Clad

```
double _d_k = 0;  
double k = 2.5;  
x += k;  
double f_return = x;  
goto _label0;  
_label0:  
* _d_x += 1;  
{  
    double _r_d0 = * _d_x;  
    * _d_x += _r_d0;  
    _d_k += _r_d0;  
    * _d_x -= _r_d0;  
    * _d_x;  
}
```

# Examples

Original code

```
double f(double x, double y) {  
    double k = 2.5;  
    x += k;  
    return x;  
}
```

Code differentiated by Clad

Dead code

```
double _d_k = 0;  
double k = 2.5;  
x += k;  
double f_return = x;  
goto _label0;  
_label0:  
*_d_x += 1;  
{  
    double _r_d0 = *_d_x;  
    *_d_x += _r_d0;  
_d_k += _r_d0;  
    *_d_x -= _r_d0;  
    *_d_x;  
}
```

k doesn't depend on x and y in a differentiable way!



# Examples

## Original code

```
double f(double x, double y) {  
    double k = 2 * x;  
    if (k > 2)  
        x *= 2;  
    return x;  
}
```

k doesn't influence the return value in a differentiable way!

## Code differentiated by Clad

```
double _t0;  
double _d_k = 0;  
bool _cond0;  
_t0 = x;  
double k = 2 * _t0;  
_cond0 = k > 2;  
if (_cond0)  
    x *= 2;  
double f_return = x;  
goto _label0;  
_label0:  
*_d_x += 1;  
if (_cond0) {  
    double _r_d0 = *_d_x;  
    *_d_x += _r_d0 * 2;  
    *_d_x -= _r_d0;  
    *_d_x;  
}  
{  
    double _r0 = _d_k * _t0;  
    double _r1 = 2 * _d_k;  
    *_d_x += _r1;  
}
```

# Examples

Original code

Code differentiated by Clad

```
double f(double x, double y) {  
    double k = 2 * x;  
    if (k > 2)  
        x *= 2;  
    return x;  
}
```

```
double _t0;  
double _d_k = 0;  
bool _cond0;  
_t0 = x;  
double k = 2 * _t0;  
_cond0 = k > 2;  
if (_cond0)  
    x *= 2;  
double f_return = x;  
goto _label0;  
_label0:  
*_d_x += 1;  
if (_cond0) {  
    double _r_d0 = *_d_x;  
    *_d_x += _r_d0 * 2;  
    *_d_x -= _r_d0;  
    *_d_x;  
}  
{  
    double _r0 = _d_k * _t0;  
    double _r1 = 2 * _d_k;  
    *_d_x += _r1;  
}
```

Dead code

k doesn't influence the return value in a differentiable way!



# Goals

- Find the best optimizing strategy for Clad and implementing it
- Investigate the potential of the Clang static analysis and data-flow analysis infrastructure to capture advanced optimization opportunities
- Investigate the possibility of enabling Clad in ADBench infrastructure
- Test in major workflows such as ROOT's RooFit package
- Writing new tests / documentation
- Create benchmarks to compare the efficiency with activity analysis on and off