Clad -- Automatic Differentiation in C++ and Clang

Vassil Vassilev, Princeton University

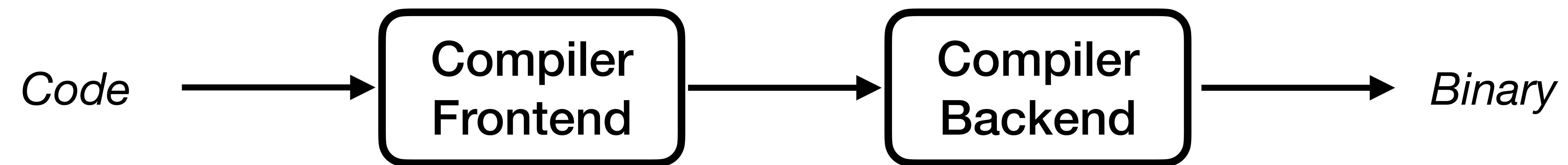
Motivation

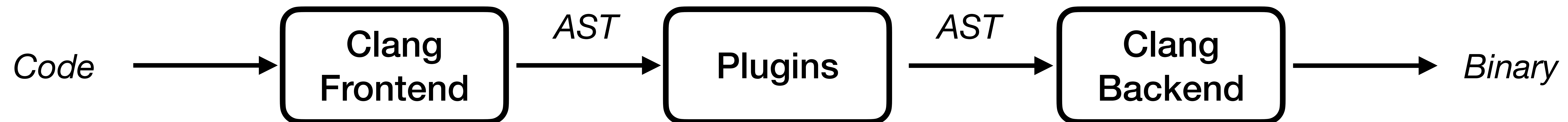
Provide automatic differentiation for C/C++ that works without code modification (including legacy code)

AD in C++. Approaches

- Challenging due to language parsing complexity. Many new features each standard. Combination of volatile vs immutable libraries.
- Classical implementation approaches:
  - Template metaprogramming
    - + Supports almost all language constructs*
    - - Requires special types for modeling dual numbers or triggering operator overloading
  - Source code transformation (mostly tools with custom parsers)
    - + Work well on existing code
    - - Hard to maintain and support all language constructs
- Such a tradeoff can be avoided and potentially give us access to even more powerful framework for AD

Typical C++ Compilation Pipeline





Clad. AD Plugin for Clang

Clad is a compiler plugin extending clang able to produce derivatives in both forward and reverse mode:

- Supports derivatives (partial and higher order), gradients, hessians and jacobians.
- Provides low-level derivative access primitives
- Allows embedding in frameworks
- Caveat — the compiler must see the source code of the target function
  - We support multiple ways to relax that but we are limited in time for this talk

Clad. Usage

The body will be generated by clad.

Result via the clad function-like wrapper object

Result via function pointer call

Result via a forward declaration

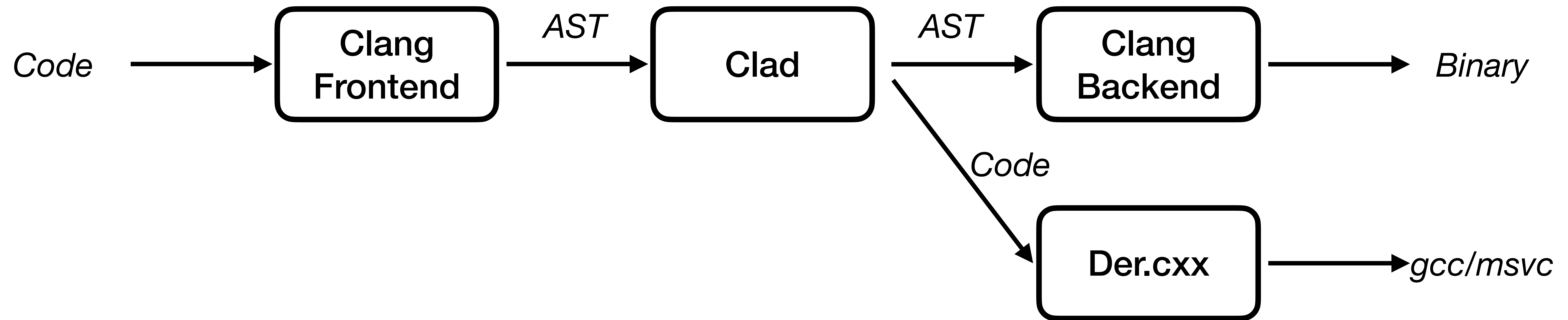
```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load
//       -Xclang libclad.so …
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }

double pow2_darg0(double);

int main() {
  auto dfdx = clad::differentiate(pow2, 0);

  // Function execution can happen in 3 ways:
  // 1) Using CladFunction::execute method.
  double res = cladPow2.execute(1);

  // 2) Using the function pointer.
  auto dfdxFnPtr = cladPow2.getFunctionPtr();
  res = cladPow2FnPtr(2);

  // 3) Using direct function access through fwd declaration.
  res = pow2_darg0(3);
  return 0;
}
```

Clad. Beyond Clang

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

```
FunctionDecl f 'double (double)'
|-ParmVarDecl x 'double'
`-CompoundStmt
  `-ReturnStmt
    `-BinaryOperator 'double' '*'
      |-ImplicitCastExpr 'double' <LValueToRValue>
      | `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
      `-ImplicitCastExpr 'double' <LValueToRValue>
        `-DeclRefExpr 'double' lvalue ParmVar 'x' 'double'
```

```
FunctionDecl 0x7f7f801dbff8 <<invalid sloc>> <invalid sloc> f_darg0 'double (double)'
|-ParmVarDecl 0x7f7f801dc090 <<invalid sloc>> <invalid sloc> used x 'double'
`-CompoundStmt 0x7f7f801dc3d0 <<invalid sloc>>
  |-DeclStmt 0x7f7f801dc190 <<invalid sloc>>
  | `-VarDecl 0x7f7f801dc118 <<invalid sloc>> <invalid sloc> used _d_x 'double' cinit
  |   `-ImplicitCastExpr 0x7f7f801dc178 <<invalid sloc>> 'double' <IntegralToFloating>
  |     `-IntegerLiteral 0x7f7f801dc0f8 <<invalid sloc>> 'int' 1
  `-ReturnStmt 0x7f7f801dc398 <<invalid sloc>>
    `-BinaryOperator 0x7f7f801dc318 <<invalid sloc>> 'double' '+'
      |-BinaryOperator 0x7f7f801dc298 <<invalid sloc>, T.cpp:3:32> 'double' '*'
      | |-ImplicitCastExpr 0x7f7f801dc268 <<invalid sloc>> 'double' <LValueToRValue>
      | | `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
      | `-ImplicitCastExpr 0x7f7f801dc280 <col:32> 'double' <LValueToRValue>
      |   `-DeclRefExpr 0x7f7f801dc208 <col:32> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
      `-BinaryOperator 0x7f7f801dc2f0 <col:30, <invalid sloc>> 'double' '*'
        |-ImplicitCastExpr 0x7f7f801dc2c0 <col:30> 'double' <LValueToRValue>
        | `-DeclRefExpr 0x7f7f801dc1d0 <col:30> 'double' lvalue ParmVar 0x7f7f801dc090 'x' 'double'
        `-ImplicitCastExpr 0x7f7f801dc2d8 <<invalid sloc>> 'double' <LValueToRValue>
          `-DeclRefExpr 0x7f7f801dc1a8 <<invalid sloc>> 'double' lvalue Var 0x7f7f801dc118 '_d_x' 'double'
```




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

AD on the fly

Some domains benefit from supporting derivatives of user code. Interpretative languages such as python can provide that easily. C++ has cling — an interactive, llvm-based C++ interpreter.

AD on the fly

The demo shows cling use clad as a plugin to produce a derivative on the fly

AD as a Service

- A service capable of running AD on a given code at program’s runtime
- Runs embedded in your framework code with your favorite compiler

AD as a Service

Declare the code to the interpreter

Move the interpreter result to the compile world

Cast and execute

```
#include <cling/Interpreter/Interpreter.h>
#include <cling/Interpreter/Value.h>

// Derivatives as a service.

void gimme_pow2dx(cling::Interpreter &interp) {
  // Definitions of declarations injected also into cling.
  interp.declare("double pow2(double x) { return x*x; }");
  interp.declare("#include <clad/Differentiator/Differentiator.h>");
  interp.declare("auto dfdx = clad::differentiate(pow2, 0);");

  cling::Value res; // Will hold the evaluation result.
  interp.process("dfdx.getFunctionPtr();", &res);

  using func_t = double(double);
  func_t* pFunc = res.getAs<func_t*>();
  printf("dfdx at 1 = %f\n", pFunc(1));
}

int main(int argc, const char* const* argv) {
 std::vector<const char*> argvExt(argv, argv+argc);
  argvEx.push_back("-fplugin=etc/cling/plugins/lib/clad.dylib");
  // Create cling. LLVMDIR is provided as -D during compilation.
  cling::Interpreter interp(argvExt.size(), &argvExt[0], LLVMDIR);
  gimme_pow2dx(interp);
  return 0;
}
```

```
./clad-demo
dfdx at 1 = 2.000000
```

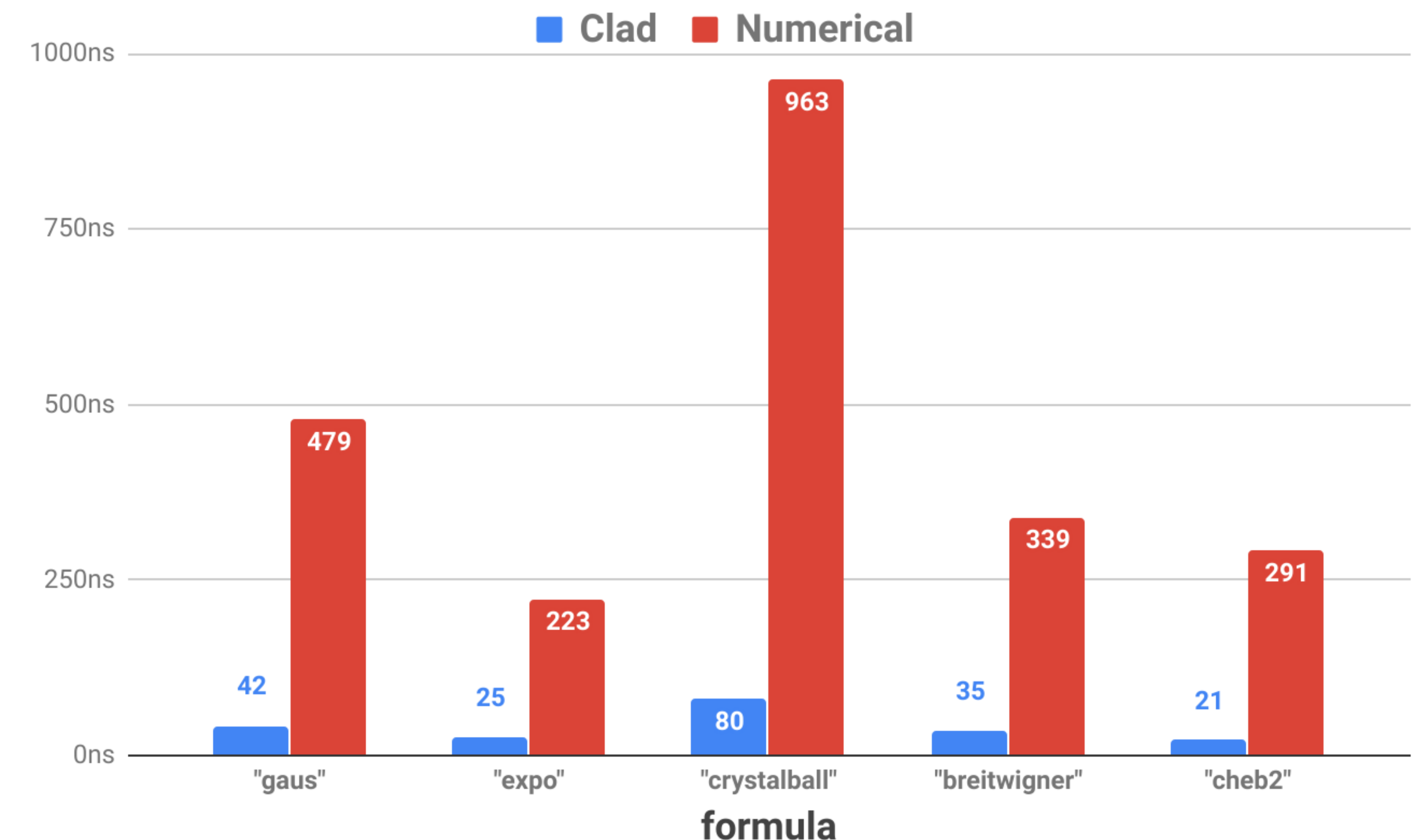
Result from running the clad-demo binary

Applications in High-energy physics

```
TF1* h1 = new TF1("f1", "formula");
TFormula* f1 = h1->GetFormula();
f1->GenerateGradientPar(); // clad
```

```
// clad
f1->GradientPar(x, result);
// numerical
h1->GradientPar(x, result);
```

- **gaus: Npar = 3**
- **expo: Npar = 2**
- **crystalball: Npar = 5**
- **breitwigner: Npar = 5**
- **cheb2: Npar = 4**



Future

- Develop error estimation framework — at compile time and at runtime
- Retarget code on GPGPU — OpenCL and/or CUDA
- Support functor objects
- Make the derivation process explicitly configurable

Thank you.

https://github.com/vgvassilev/clad

Backup. Clad Details

Clad. Implementation

```
// clang -Xclang -add-plugin -Xclang clad -Xclang -load
//       -Xclang libclad.so …
// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }

// The body will be generated by clad.
double pow2_darg0(double);

int main() {
  auto dfdx = clad::differentiate(pow2, 0);

  // Function execution can happen in 3 ways:
  // 1) Using CladFunction::execute method.
  double res = cladPow2.execute(1);

  // 2) Using the function pointer.
  auto dfdxFnPtr = cladPow2.getFunctionPtr();
  res = cladPow2FnPtr(2);

  // 3) Using direct function access through fwd declaration.
  res = pow2_darg0(3);
  return 0;
}
```

Clang creates a CladObject with pow2,
Clad swaps its content to the generated pow2_darg0

Clad. Debugging

```
double sum(double* p, int dim) {
  double r = 0.0;
  for (int i = 0; i < dim; i++)
    r += p[i];
  return r;
}
```

clad::CladFunction::dump

```
void sum_grad_0(double *p, int dim, double *_result) {
    double _d_r = 0;
    unsigned long _t0;
    int _d_i = 0;
    clad::tape<int> _t1 = {};
    double r = 0.;
    _t0 = 0;
    for (int i = 0; i < dim; i++) {
        _t0++;
        r += p[clad::push(_t1, i)];
    }
    double sum_return = r;
    _d_r += 1;
    for (; _t0; _t0--) {
        double _r_d0 = _d_r;
        _d_r += _r_d0;
        _result[clad::pop(_t1)] += _r_d0;
        _d_r -= _r_d0;
    }
}
```

Clad. Custom Derivatives

```
double f(double x) { ... ; y = std::sin(x); ...  }
namespace custom_derivatives {
  namespace std {
    double sin_darg0(double x) { return ::std::cos(x); }
  }
}

// Detected by clad and swapped.
double f_darg0(double x) { ... ; dy = custom_derivatives::std::sin(x); ...   }
```