IRIS-HEP

Proposal

Add support for custom types in Clad with a focus on the Softsusy library.

By

Parth Arora

Table of Contents

1. Abstract

Clad is a Clang plugin that can differentiate mathematical functions represented as C++ functions using automatic differentiation coupled with analysing and transforming the clang abstract syntax tree. Clad is an emerging automatic differentiation tool that holds great potential. It already supports many exciting features such as array differentiation, differentiating functors and lambda expressions, built-in error estimation framework and so much more.

A few crucial things that are holding Clad down are:

- Not supporting user-defined (custom) types.
- Not supporting C++ STL.
- Supporting only a subset of C++ syntaxes.
- Not supporting calls to member functions in both the forward mode AD and the reverse-mode AD.
- Limited support for calls to ordinary functions in the reverse mode AD and the forward mode AD.

This project majorly focuses on realising clad potential by making it compatible with and battle testing it on Softsusy and Eigen libraries codebases. This project assumes that all definitions of the functions that require differentiation are provided in the same file or header-only header files, such that no linking is required.

User-defined types in C++ help to make code more readable and maintainable. Many user-defined programs and almost every major library uses user-defined types. Thus it is very crucial for clad to support differentiating user-defined types. In light of this, the first goal of the project is to add support for differentiating user-defined types in clad.

Clad currently does not support several C++ syntax constructs. Many of these are essential and are very well used in day-to-day programming such as *break* and *continue* statements that are currently not supported in clad reverse-mode AD. The second goal of the project is to battle test clad on Softsusy and Eigen libraries codebases to find and add support for most of the missing syntax as well as to improve support for differentiating function calls.

2. Deliverables

- Add support for differentiating user-defined (custom) types.

- Add support for missing C++ syntax constructs.

  A non-exhaustive list of missing C++ syntax constructs:

  - *break* and *continue* statements in reverse mode AD.
  - *switch* statement in reverse mode AD.
  - *new and delete* statements.
  - Recursive functions in the forward mode AD.
  - Range loops
  - Iterators
  - Lambda expression calls

- Improve support for differentiating calls to non-member functions.

- Add support for differentiating more varieties of function signatures.

- Add support for differentiating calls to member functions and overloaded operators

- Extend the test and benchmark coverage.

3. Project Details

Add support for differentiating user-defined types.

```
UserDefinedMatrixType fn(UserDefinedMatrixType matrix, UserDefinedVectorType vec) {
  return (matrix + 2*vec)*3;
}
```

Example of function using user-defined types.

After adding support for differentiating user-defined types, we will be able to use them as the function return type, function parameter types and inside the function body.

Some of the challenges and implementation details are described below:

Derived types of user-defined types.

In the derived type, we need a separate variable to store the derivative of each member variable. If we differentiate a variable of type *double* with respect to a member variable of type *double,* then we will require a variable of type *double* to store the derivative. Similarly, if we are differentiating a variable of type *UserDefinedVectorType* with respect to a member variable of type *double,* then we will require a variable of type *UserDefinedVectorType* to store the derivative and so on.

For each class and the type of output variable, we will need a new type to store the derivative of the output variable with respect to the class object. The basic idea is that each primitive type should be replaced by the type of the output parameter.
For example, the derived type for storing the result of differentiating a user-defined object of type *RealType* with respect to an object of user-defined type *UserDefinedVectorType*, is described below:

UserDefinedVectorType

```
class UserDefinedVectorType {
  double* m_vec = nullptr;
  double m_OtherMem1;
  double m_OtherMem2;
  ...
  ...
};
```

__clad_UserDefinedVectorType_DerivedType_Real

```
class __clad_UserDefinedVectorType_DerivedType_Real {
  Real* m_vec = nullptr;
  Real m_OtherMem1;
  Real m_OtherMem2;
  ...
  ...
}
```

__clad_UserDefinedVectorType_DerivedType_Real is used to store derivatives of a *Real* object type wrt *UserDefinedVectorType* object.

Creating the derived types of user-defined types.

We can either decide to create derived types automatically using Abstract Syntax Tree (AST) transformation through Clad or let the user create the derived type manually.

Creating derived types automatically through clad has a few problems and challenges that need to be addressed.

Until now, in clad, we have always created derived types of functions at compile time using C++ metaprogramming because creating derived types through AST transformation is error-prone and plugging in the newly created type in the abstract syntax tree is a non-trivial task since it will involve examining and modifying the surrounding AST to keep the semantics intact.

Another challenge of automatically creating derived types through Clad is that users need to pass an address of the object in reverse mode AD functions to store the derivatives.

Now, if the derived type is created automatically through clad, then the users would not be able to access the derived type to create the object that should store the derivative.

Please note that we cannot create derived types of classes through metaprogramming because there is no way to traverse over class members.

Considering these problems, we have few options to resolve them.

- **The user manually defines the derived type.**

  In this idea, the user will manually create the derived type for the user-defined type and clad will use that derived type wherever it requires the derived type of that user-defined class.

  This method puts more work on the user and thus it is less desirable.

- **User manually only forward-declares the derived type and clad automatically defines the type.**

  In this idea, users will need to manually forward-declare all the derived types. On encountering the forward declarations clad will automatically generate the derived types. This idea is better because it reduces the work from the user yet solves most of the challenges that were discussed above.

  For example, if the user writes the code that is shown below, then clad will automatically generate the definition of the class

_clad_UserDefinedVectorType_DerivedType_Real.

```
class UserDefinedVectorType {
  double* m_vec = nullptr;
  double m_OtherMem1;
  double m_OtherMem2;
  ...
  ...
};

class __clad_UserDefinedVectorType_DerivedType_Real;
```

Type of derived member functions

Since member functions usually call other member functions and implicitly have access to the *this* pointer to access the class object. They also need to have access to the derivative of *this* object with respect to the output parameter.
For example, consider a member function such as shown below:

```
class UserDefinedVectorType {
  std::vector<double> m_vec;
  ... other data members ...

  public:
  void reverse() {
    std::reverse(m_vec.begin(), m_vec.end());
  }
};
```

If the reverse member function is called in some other function where an object of *UserDefinedVectorType* is used and the output parameter is of *Real* type. Then we will need to create a new member function such as *reverse_grad* that is as follows:

```
void reverse_grad(__clad_UserDefinedVectorType_DerivedType_Real& derived_obj) {
  ...
  ...
}
```

Please note that we may need to create a new derived member function for each output variable since a different derived type is created for a different output variable.

The rest of the implementation details of adding support for differentiating user-defined types are relatively straightforward and will be inspired by the current implementation of deriving ordinary variables in Clad.

Add support for missing C++ syntax constructs.

Adding support for many of the missing C++ syntax constructs is straightforward to implement and depends greatly on the particular syntax. For some other syntax, adding support is much more interesting and complicated.

Here, I will discuss implementation details of adding support for some of the syntaxes.

Adding support of *break* and *continue* statement in the reverse mode loops

break and *continue* statements abruptly modify the control flow of the code. To correctly compute the derivative, Clad needs to keep track of which statements were executed in the forward iteration of the loop so that clad can execute corresponding derived statements in the associated reverse iteration of the loop. For this, clad needs to store which *break/continue* statement was hit in which loop iteration.

Since Clad can only determine which *break/continue* will be hit at runtime. We need a way to change the control flow of code at runtime depending on which *break/continue* was hit. To handle this we can enclose the entire body of the loop in the *switch* statement and associate a case label with each *break/continue statement.* Here the *switch* statement will work as a goto statement whose associated label can be specified at runtime. Using this strategy, we will store which *break/continue* statement was hit in a clad tape. Since the clad tape is based on the stack data structure, it will allow us to retrieve information regarding which *break/continue* was hit in the reverse order (starting from the last iteration of the loop) and that's exactly what we want, thus it will make implementation relatively easier.

An example of this transformation is described below:

```
while (choice--) {
  if (choice > 3) {
    res += i;
    continue;
  }
  if (choice > 1) {
    res += j;
    break;
  }
}
```

Original loop

```
while (choice--) {
  _t0++;
  bool _t1 = choice > 3;
  {
    if (_t1) {
      res += i;
      {
        clad::push(_t3, 1UL);
        continue;
      }
    }
    clad::push(_t2, _t1);
  }
  bool _t4 = choice > 1;
  {
    if (_t4) {
      res += j;
      {
        clad::push(_t3, 2UL);
        break;
      }
    }
    clad::push(_t5, _t4);
  }
  clad::push(_t3, 3UL);
}
```

Forward block

```
while (_t0) {
  switch (clad::pop(_t3)) {
    case 3UL:;
      if (clad::pop(_t5)) {
        case 2UL:;
          {
            double _r_d1 = _d_res;
            _d_res += _r_d1;
            *_d_j += _r_d1;
            _d_res -= _r_d1;
          }
      }
      if (clad::pop(_t2)) {
        case 1UL:;
          {
            double _r_d0 = _d_res;
            _d_res += _r_d0;
            *_d_i += _r_d0;
            _d_res -= _r_d0;
          }
      }
  }
  _t0--;
}
```

Reverse block

Adding support for differentiating *for range* loops

Conceptually, the implementation of adding support for differentiating *for range* loops will be very similar to the current implementation of differentiation of *for* and *while* loops. Therefore, they will be used as an inspiration for adding support for differentiating *for range* loops.

During the project, clad will be battle-tested on codebases of C++ STL, Softusy and Eigen libraries. It is expected much more missing syntax will be discovered during this.

Improve support for differentiating calls to ordinary functions.

Currently, the forward mode AD does not support differentiating multiple arguments function calls. Please see issue #168 for more details.
This can be relatively easily solved by calculating the gradient of the callee function and then using the computed gradient to correctly calculate derivatives.

And also neither the forward mode AD nor the reverse mode AD support differentiating function calls with pointers as function arguments.

This can also be relatively easily solved by implementing a general dereference operation to use whenever a pointer is found. The general implementation will aim to allow differentiating an arbitrary number of dereferencing operators with one variable (**var, **var, ***var,* and so on*)*.

If we are differentiating a function call that takes reference variables as arguments, then in reverse mode, we may need to consider that the callee function is a pure function (it always gives the same value of output for the same values of arguments). I need to research more about the handling of reference variables as parameters before saying anything about its implementation.

Add support for differentiating calls to member functions and overloaded operators.

For the most part handling of calls to member functions will be similar to the handling of calls to ordinary functions, with one big exception, member functions will have an extra parameter, to store the derivative of the *this* pointer object. I currently need to research more about the implementation of ordinary functions and member functions.

Internally calls to overloaded operators are handled in the same way as calls to member functions, therefore, implementation for adding support for differentiating calls to the overloaded operators will be able to reuse components defined for adding support for differentiating calls to member functions.

Add support for differentiating more varieties of function signatures.

Many of the functions in the Softsusy library have a different signature than what clad supports.

```
void fn(DoubleMatrix& m1, DoubleMatrix& m2, DoubleMatrix& m3, const sBrevity a);
```

Example of one such function

This function has 3 outputs, *m1, m2* and *m3*. Currently, there is no function in Clad that can calculate the Jacobian matrix in which *m1, m2* and *m3* are differentiated with respect to *a.*

Therefore we need to add support for differentiating more varieties of function signatures in clad.

Implementation of support for differentiating more varieties of function signatures will be inspired by how Clad handles differentiation of currently supported function signatures. I need to research more about it before saying anything regarding its implementation details.

Additional Work

Additionally, I want to work on the following areas as well to improve Clad.

Add automated assert-based testing of clad automatic differentiation library using clad numerical differentiation library.

This work focuses on making Clad testing framework more robust. Since clad is an open-source project, and like most open-source different people will work on different parts of the project over the years. A good and robust testing framework will make it easier and faster to find bugs.

For this we will first need to modify clad numerical differentiation functionality to work standalone, currently, it only works as a fallback to automatic differentiation.

This functionality can be implemented either as a plugin for the *ForwardModeVisitor* and the *ReverseModeVisitor*. It will modify the generated derived function to include assert statements at the end to verify that results from both automatic differentiation and numerical differentiation are consistent with each other, and of course, we will need to decide a satisfiable value of accepted(or allowed) error since numerical differentiation results have a precision loss.

Make generated source file through *-fgenerate-source-file* standalone usable

The goal of this work is to make the generated source file using *-fgenerate-source-file* error-free and standalone usable. This requires at least the following:

- Issue #115 needs to be fixed. This issue leads to the generation of multiple definitions of the same function in the generated source file thus leading to redefinition errors.

- When we are writing derived member functions to the file, we are only writing derived member functions and not the associated class. In the derived member functions, we are using *this* pointer and class member variables which are causing a compile-time error. Therefore we should print the whole class along with derived member functions.

4. Schedule of Deliverables

- There are 5 major milestones:
    1. Adding support for user-defined types.
    2. Add support for missing C++ syntax and constructs.
    3. Improve support for ordinary function calls and add support for member function calls
    4. Battle-test clad using Softsusy and Eigen library.
    5. Complete work specified under additional work in project details.

- If more things come up or plan change, they can also be incorporated

Date	**Work**
Nov 5 - Nov 19	Start working on adding initial support of differentiating user-defined types.
Nov 19 - Dec 3	Complete initial support of differentiating user-defined types; add tests and documentation for the same.
Dec 3 - Dec 17	Improve support for differentiating calls to ordinary functions in the forward mode AD.
Dec 17 - Dec 31	Improve support for differentiating calls to ordinary functions in the reverse mode AD.
Dec 31 - Jan 14	Add support for differentiating calls to member functions in reverse mode AD.
Jan 14 - Jan 28	Add support for differentiating calls to member functions in the forward mode AD
Jan 28 - Feb 11	Add support for differentiating overloaded operators
Feb 11 - Feb 25	Add new signatures of clad differentiation functions as is required for Sofsusy
Feb 25 - Mar 11	Start adding support for missing syntax such as *new, delete, for range loops, switch statements,* arbitrary pointer dereference.
Mar 11 - April 1	Battle test clad using Softsusy and Eigen library; add support of any missing syntax.

Apr 1 - Apr 8	Add automatic assert-based testing of reverse-mode AD using clad numerical differentiation library
Apr 8 - Apr 22	Make generated source file through *-fgenerate-source-file* standalone usable
Apr 22 - May 5	Buffer-time. Test and submit final code and project summaries. Prepare documentation and remove any bugs

5. Obligations

I will be available to work part-time (a minimum of 25 hours per week) during the project period.

6. General Notes

I believe that communication is a vital aspect of any project and to ensure that the status of the project is communicated properly, I will contact my mentors *at least* once every 3 days to update them on how the work is progressing, discuss current problems and ask for their suggestions for the problems at hand.

7. About Me

Name: Parth Arora

Email: partharora99160808@gmail.com

Github: parth-07

Linkedin: parth-r07

Resume: link

Phone number: +91 9354826906

Time Zone: +5:30 GMT

University Name: University School of Information, Communication and Technology, GGSIPU, New Delhi, India.

Course: Bachelor of Technology in Computer Science Engineering

CGPA: 8.5

Current Year: 4th