Tools for Algorithmic Differentiation

Keywords: *Algorithmic Differentiation, Clang, LLVM, Source Analysis, Source Transformation*

Alexander Hück
alexander.hueck@tu-darmstadt.de

Office: S1|22 Room 412
Alexanderstraße 2
64287 Darmstadt
Tel. 06151 16-7 55 77

Date: 18th June, 2020

Introduction

This document gives a brief introduction to Algorithmic Differentiation (AD, [1]) and work related to it at the Scientific Computing Group.

Algorithmic Differentiation (AD)

AD relies on overloading operators in C++ to calculate derivatives.

Operator Overloading

C++ allows for the customization of the semantics of common (elemental) operators by introducing user-defined types, called operator overloading (overloading) [2]. Using overloading, the AD type $\widetilde{T}$ replaces the floating-point type T (`double`, `float`) of the target code. It overloads all relevant operations and mathematical functions. For each invoked overloaded operation, a new (temporary) object of type $\widetilde{T}$ with updated derivative values based on the executed operator is returned,

$$\widetilde{T} \circ \widetilde{T} \mapsto \widetilde{T} \,.$$

The operator $\circ$ represents the operation that is applied to combine the two values of type $\widetilde{T}$, e.g., a multiplication. This is done with, e.g., an additional derivative value that is encapsulated in $\widetilde{T}$ along the primal (original) value of type T.

Example

The implementation of a (naive) overloading AD tool is shown in Figure 1.

```
class adouble  {
public:
  double p; // primal value
  double d; // derivative value

  adouble(double primal, double deriv=0.0) : p(primal), d(deriv) { }
  // Multiplication operator overload:
  adouble operator*(const adouble& other) {
      return adouble(/*primal =*/ p*other.p,
                     /*deriv. =*/ p*other.d + d*other.p);
  }
};
```

Figure 1: A minimal AD overloading implementation called `adouble`. The multiplication is overloaded, it returns a new object with updated derivatives. Other operators are implemented equivalently.

The derivative computation is done by changing the type of the target code and executing it with the new type with a prior seeding, see Figure 2.

Alexander Hück
alexander.hueck@tu-darmstadt.de

Office: S1|22 Room 412
Alexanderstraße 2
64287 Darmstadt
Tel. 06151 16-7 55 77

Date: 18th June, 2020

```
double foo(double x) {
  return x*x*x;
}
void bar() {
  double x = 3.0;
  double y = foo(x);
  std::cout << y
            << std::endl;
}
```

```
adouble foo(adouble  x) {
  return x*x*x;
}
void bar() {
  adouble x = 3.0;
  x.d=1.0;  // initial deriv. of x
  adouble y = foo(x);
  std::cout << y.p
            << y.d
            << std::endl;
}
```

Figure 2: On the left, the original target code is shown. On the right, the **double** type is replaced by the **adouble** type

Complications

While the application of AD with overloading looks straightforward, the augmentation can lead to, e.g., many compiler errors and often requires substantial effort to integrate into existing code bases. For instance, the **union** class type may cause problems with the user-defined AD type [3]. Hence, the development efforts concentrate on compiler tooling, which aims to make the application and use of AD more straightforward.

Tasks

We are interested in students who want to help develop tools powered by compiler technology [4] to assist with the application of algorithmic differentiation. This pertains to, e.g.,

(a) static code analysis,

(b) memory tracking for correctness checks, or

(c) source transformations.

Requirements

- Strong fundamentals w.r.t. modern C++
- Knowledge of the CMake build system
- Experience with the Clang and LLVM compiler framework

References

[1] http://www.autodiff.org

[2] https://en.cppreference.com/w/cpp/language/operators

[3] §9.5-1, C++03 Standard

[4] https://clang.llvm.org/docs/LibTooling.html