Recursive compile-time differentiation

Handling nested functions

Dominic Jones

July 2020

Introduction

- The compile time differentiation of a C++ function which only calls built-in functions has been demonstrated in other talks
- This talk describes the approach for differentiating a function which may call any other function
- The approach produces very efficient code, and compiles relatively quickly
- Siemen's Simcenter STAR-CCM+ simulation software has an implementation of this approach, and is used to differentiate the Spalart Allmaras turbulence model, among other things

Perfect forwarding

```
template<class OP, class T>
struct Unary { Unary(T &&v) {} };
```

```
class Sqrt;

template<class T>
auto sqrt(T &&v)
{
  return Unary<Sqrt, T>(std::forward<T>(v));
}
```

```
float const a0{1};
auto a1 = sqrt(a0);         // Unary<Sqrt, float const &>

float b0{1};
auto b1 = sqrt(b0);         // Unary<Sqrt, float &>

auto c1 = sqrt(float{1});   // Unary<Sqrt, float>
```

Built in functions

Hypotenuse

$$r = \sqrt{a^2 + b^2}$$

```
float a = 3;
float b = 4;
float r;

{
  float d = a*a + b*b;
  r = sqrt(d);
}

std::cout << r << std::endl; // r = 5
```

Primal of Hypotenuse

$$r = \sqrt{a^2 + b^2}$$

```
float a = 3;
float b = 4;
float r;

auto constexpr mode = DrvMode::PRIMAL;

Drv<mode, float>  a_{a};
Drv<mode, float>  b_{b};
Drv<mode, float&> r_{r};

{
  EDrv<mode, float> d = a_*a_ + b_*b_;
  r_ = drv::sqrt(d);
}

std::cout << r << std::endl; // r = 5
```

Tangents of Hypotenuse

$$\frac{dr}{da}$$

```
float a = 3, a_drv = 1; // w.r.t. 'a'
float b = 4, b_drv = 0;
float        r_drv;

auto constexpr mode = DrvMode::TANGENT;

Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  EDrv<mode, float> d = a_*a_ + b_*b_;
  r_ = drv::sqrt(d);
}

std::cout << r_drv << std::endl; // dr/da = 0.6
```

Tangents of Hypotenuse

$$\frac{dr}{db}$$

```
float a = 3, a_drv = 0;
float b = 4, b_drv = 1; // w.r.t. ‘b’
float        r_drv;

auto constexpr mode = DrvMode::TANGENT;

Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  EDrv<mode, float> d = a_*a_ + b_*b_;
  r_ = drv::sqrt(d);
}

std::cout << r_drv << std::endl; // dr/db = 0.8
```

Adjoint of Hypotenuse

$$\begin{bmatrix} \frac{dr}{da} & \frac{dr}{db} \end{bmatrix}^T$$

```
float a = 3, a_drv = 0;
float b = 4, b_drv = 0;
float        r_drv = 1; // w.r.t. 'r'

auto constexpr mode = DrvMode::ADJOINT;
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

// scope is required
{
  EDrv<mode, float> d = a_*a_ + b_*b_;
  r_ = drv::sqrt(d);
}

std::cout << a_drv << std::endl; // dr/da = 0.6
std::cout << b_drv << std::endl; // dr/db = 0.8
```

User defined functions

As a subrountine

```
template<DrvMode::Option mode> void
hyp(Drv<mode, float> const &a,
    Drv<mode, float> const &b,
    Drv<mode, float&> r)
{
  EDrv<mode, float> d = a*a + b*b;
  r = drv::sqrt(d);
}
```

```
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

hyp(a_, b_, r_); // subroutine style (not much use...)
```

As a function

```
struct Hyp {
  template<DrvMode::Option mode> static void
  evaluate(Drv<mode, float> const &a,
           Drv<mode, float> const &b,
           Drv<mode, float&> r)
  {
    EDrv<mode, float> d = a*a + b*b;
    r = drv::sqrt(d);
```

```
template<class E0, class E1> auto hyp(E0 &&e0, E1 &&e1) ->
DrvVariadicNode<find_DrvMode<E0, E1>::result(), // mode
                decltype(primal(e0 + e1)),      // float
                ScopedExprBinding<Hyp>, E0, E1>
{
  return {std::forward<E0>(e0), std::forward<E1>(e1)};
```

```
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

r_ = hyp(a_, b_); // functional style (very useful!)
```

Continuation with functions

```
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float>  b_{b, b_drv};
Drv<mode, float&> r_{r_drv};

{
  EDrv<mode, float> r  = hyp(a_, b_);
  EDrv<mode, float> r2 = drv::pow(r, 2);
  r_ = r2;
}
```

- `hyp` can be used just like any built-in function, such as `drv::pow`

Beyond the basics

Multiple results

```
MeanSd::
evaluate(...,
         Drv<mode, std::tuple<float, float> &> r)
{
  EDrv<mode, float> mean = a + b / 2;
  EDrv<mode, float> sd2 = drv::pow(a - mean, 2) +
                          drv::pow(b - mean, 2);
  r.at<1>() = drv::sqrt(sd2 / 2);
  r.at<0>() = mean;
```

```
{
  EDrv<mode, std::tuple<float, float>> r = mean_sd(a_, b_);
  EDrv<mode, float> mean = r.at<0>();
  EDrv<mode, float> sd = r.at<1>();
  ...
```

- `mean_sd` packages outputs with `std::tuple` and accesses them with `result.at<I>()`

Passive variables

```
Drv<mode, float>  a_{a, a_drv};
Drv<mode, float&> r_{r_drv};

{
  EDrv<mode, float> r  = hyp(a_, b);      // ‘b’ is passive
  EDrv<mode, float> r2 = drv::pow(r, 2); // ‘2’ is passive
  r_ = r2;
}
```

- At least one parameter of every function needs to be an active variable or expression (i.e. a `Drv<>` or `EDrv<>`)

`l-value` types

Named approach (using heap & stack)

```
{
  EDrv<mode, float> d = a*a + b*b;
  r = drv::sqrt(d);
}
```

- `EDrv<>` doesn't know the type of the expression: `a*a + b*b`
- In order for `EDrv<>` to make a copy of the expression so as to evaluate its adjoint during destruction, `OpaqueObjectManager` is used
- The manager provides a stack buffer. If the expression is larger than the buffer then the heap is used

Named approach (using stack only)

```
{
  SDrv<mode, float> d = a*a + b*b;
  r = drv::sqrt(d);
}
```

- `SDrv<>` doesn't know the type of the expression: `a*a + b*b`
- In order for `SDrv<>` to make a copy of the expression so as to evaluate its adjoint during destruction, `OpaqueObjectManager` is used
- The manager provides a stack buffer. If the expression is larger than the buffer then there is a `static_assert` during compilation

auto approach

```
{
  auto d = edrv(a*a + b*b);
  r = drv::sqrt(d);
}
```

- `auto` knows the type of the expression: `a*a + b*b`
- `auto` holds a copy of the expression so as to evaluate its adjoint during destruction
- `auto` and `edrv` go together, rather like `std::unique_ptr` and `std::make_unique`
- This approach produces the most efficient code

Overview

- The idea is to be able to annotate original code in order to generate its derivative
- Code must be ‘pure functional’, i.e. all variables ought to be `const` qualified
- User defined primitives supported, like `Vector<N,T>`, `Tensor<N,T>`
- Virtual functions are supported
- `auto` return type is supported (instead of `EDrv<>`)

Recursive compile-time differentiation

Handling nested functions

Dominic Jones

July 2020