Compile time adjoint in C++

22^{nd} EuroAD Workshop, Imperial College

Dominic Jones

`github.com/DominicJones`

July 2019

Overview

1. **Preliminaries:** resources, basic ideas, etc
2. **Mapping indices:** separate tree structure and its data
3. **Language extensions:** functionality that would be very helpful

Preliminaries

Zero or more

```
// no state, but instantiable
template<class... T> struct list {};
```

```
// a more useful form...
template<class T, T...> struct seq {};
```

```
// _the_ operation
template<class L> struct front;

template<template<class...> class L, class T1, class... T>
struct front<L<T1, T...> >
{
  using type = T1;
};
```

Simple C++11 metaprogramming

With variadic templates, parameter packs and template aliases

Peter Dimov, 26.05.2015

I was motivated to write this after I read Eric Niebler's thought-provoking Tiny Metaprogramming Library article. Thanks Eric.

C++11 changes the playing field

The wide acceptance of Boost.MPL made C++ metaprogramming seem a solved problem. Perhaps MPL wasn't ideal, but it was good enough to the point that there wasn't really a need to seek or produce alternatives.

C++11 changed the playing field. The addition of variadic templates with their associated parameter packs added a compile-time list of types structure directly into the language. Whereas before every metaprogramming library defined its own type list, and MPL defined several, in C++11, type lists are as easy as

```
// C++11
template<class... T> struct type_list {};
```

and there is hardly a reason to use anything else.

Template aliases are another game changer. Previously, "metafunctions", that is, templates that took one type and produced another, looked like

```
// C++03
template<class T> struct add_pointer { typedef T* type; };
```

and were used in the following manner:

Compressible N-S adjoint: auto. diff. at field loop iteration level

Differentiate a function *losslessly*

Parity preserving transform

write something like this . . .

```
fn(A const &a, B const &b,
   R &r)
{
  auto c0 = 7;
  auto c1 = 9;

  auto t0 = a * b;    // 1
  auto t1 = c0 + t0; // 2
  auto t2 = c1 + t0; // 3

  r  = t1 / t2;      // 4
}
```

to implement something like this

```
fn(A const &a, B const &b,
   R &r)
{
  auto c0 = 7;
  auto c1 = 9;

  auto t0 = a * b;    // 1
  auto t1 = c0 + t0; // 2
  auto t2 = c1 + t0; // 3

  // somehow add this...
  t1.d += (1/t2)     * r.d; // 4'
  t2.d -= (t1/t2^2) * r.d; // 4'
  t0.d += t2.d;             // 3'
  t0.d += t1.d;             // 2'
  a.d  += b * t0.d;         // 1'
  b.d  += a * t0.d;         // 1'
}
```

Observations

1. Given a pure-functional algorithm, differentiate it
2. Implement the transpose of the chain of derivatives (the adjoint)
3. The required 'extra' code is in the reversed sequence of the original and the data flow is reversed
4. Eager and lazy evaluation: ctor-dtor pairs?

Two hurdles

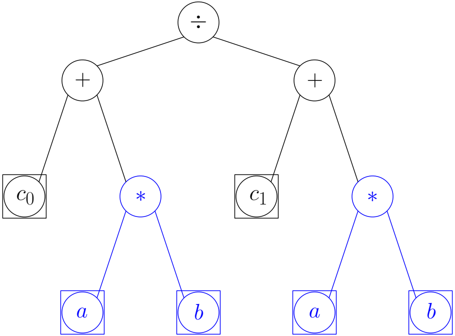
1. Dealing with duplicate nodes

Eager evaluation and capture by reference?

```
fn(A const &a, B const &b,
   R &r)
{
  auto c0 = 7;
  auto c1 = 9;

  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;

  r  = t1 / t2;
}
```



2. Dealing with nested scoping

The *complete* tree, including `cm`, is needed for the transform

```
mul_dbl(A const &a, B const &b)
{
  auto cm = 2; // locally scoped
  return cm * a * b;
}

fn(A const &a, B const &b, R &r)
{
  auto c0 = 7;
  auto c1 = 9;

  auto t0 = mul_dbl(a, b);
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;

  r  = t1 / t2;
}
```

2. Dealing with nested scoping

The *complete* tree, including `cm`, is needed for the transform

```
fn(A const &a, B const &b,
   R &r)
{
  auto c0 = 7;
  auto c1 = 9;
  auto cm = 2;
  auto t0 = cm * a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;

  r  = t1 / t2;
}
```

```
fn(A const &a, B const &b,
   R &r)
{
  auto c0 = 7;
  auto c1 = 9;
  auto cm = 2;
  auto t0 = cm * a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;

  // the transform...
  t1.d += (1/t2)     * r.d;
  t2.d -= (t1/t2^2) * r.d;
  t0.d += t2.d;
  t0.d += t1.d;
  a.d  += cm * b * t0.d;
  b.d  += cm * a * t0.d;
}
```

State of affairs

1. Eager evaluation avoids duplicate branch evaluation - but lazy evaluation will also be needed
2. ‘Capture by reference’ to keep the tree small - but cannot work with nested scoping
3. ‘Capture by value’ is too inefficient - the tree will get very large very quickly
4. A monolithic tree, supporting eager and lazy evaluation, of minimal size, and impartial to scoping is required

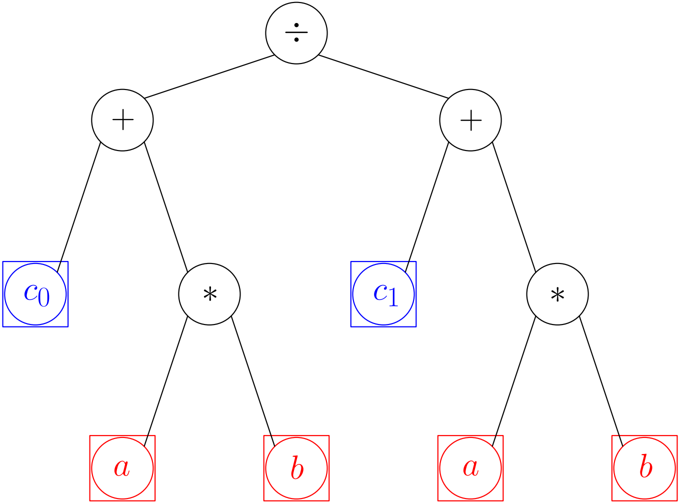
Expression tree to type list

Two kinds of data

```
auto fn(A const &a, B const &b)
{
  auto c0 = UQ(7);
  auto c1 = UQ(9);

  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;
  return t1 / t2;
}
```

Two kinds of data



UQ

```
template<std::size_t ID, typename T>
struct Unique
{
  T value;
};
```

```
// UQ
#define UQ(v) Unique<__COUNTER__, decltype(v)>{v}
```

Tree type to list of types

Generate tree with operator overloading

```
Binary<Div,
  Binary<Add, C0,
    Binary<Mul, A, B> >
  Binary<Add, C1,
    Binary<Mul, A, B> > >
```

Group hierarchically and prune duplicates (at each binary node), then flatten (at the root node)

```
{A,
 B,
 Binary<Mul, A, B>,
 Binary<Add, C0, [...]>,
 Binary<Add, C1, [...]>,
 Binary<Div, [...], [...]>}
```

Mapping nodes to data

Map `C0` and `C1` to values

```
L = {A,                                   //  0
     B,                                   //  1
     Binary<Mul, A, B>,                   //  2
     Binary<Add, C0, [...]>,              //  3  <--
     Binary<Add, C1, [...]>,              //  4  <--
     Binary<Div, [...], [...]>}           //  5
```

```
tuple<float, float> vars = {7.0, 9.0};
```

Indices of constants in ‘L’

```
IC = {3, 4}
```

Map elements in ‘L’ to storage offsets *(2 = null marker)*

```
DC = {2, 2, 2, 0, 1, 2}
```

dual

```
// input                                  output
//-------                                --------
DC_SIZE = 6      \
                  --- dual --->     DC = {2, 2, 2, 0, 1, 2}
IC = {3, 4}      /                        0  1  2  3  4  5
```

github.com/DominicJones/snippets/blob/master/Cxx/mp_functions.cpp

Mapping nodes to data

Map A and B to addresses

```
L = {A,                                      //  0  <--
     B,                                      //  1  <--
     Binary<Mul, A, B>,                      //  2
     Binary<Add, C0, [...]>,                 //  3
     Binary<Add, C1, [...]>,                 //  4
     Binary<Div, [...], [...]>}              //  5
```

```
tuple<float*, float*> args = {&a, &b};
```

Map offsets of left child nodes *(6 = null marker)*

```
IL_L = {6, 6, 0, 6, 6, 3}
```

Map offsets of right child nodes

```
IL_R = {6, 6, 1, 2, 2, 4}
```

Evaluate operator list

```
tuple<float, float>   vars = {7.0, 9.0};
```

```
tuple<float*, float*> args = {&a, &b};
```

Iterate over lists to compute primal and adjoint

```
DC   = {2, 2, 2, 0, 1, 2}
```

```
IL_L = {6, 6, 0, 6, 6, 3}
```

```
IL_R = {6, 6, 1, 2, 2, 4}
```

```
L = {A,                                    //  0
     B,                                    //  1
     Binary<Mul, A, B>,                    //  2
     Binary<Add, C0, [...]>,               //  3
     Binary<Add, C1, [...]>,               //  4
     Binary<Div, [...], [...]>}            //  5
```

Results

Case 1: nodes: 82, depth: 37, inputs: 2, constants: 12

Version	compilation	original	auto diff	manual diff
`alt::tuple`	2.2s	1x	1.25x	1.48x
`std::tuple`	4.5s	1x	1.25x	1.48x

Case 2: nodes: 331, depth: 25, inputs: 5, constants: 103

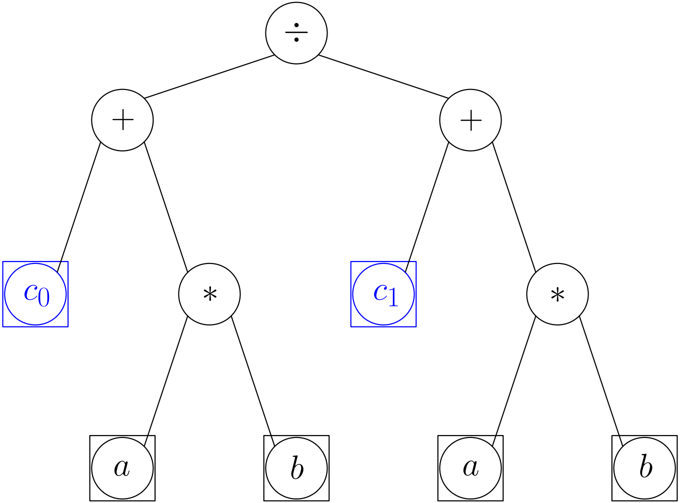
Version	compilation	original	auto diff	manual diff
`alt::tuple`	59s	1x	5.7x	1.9x
`std::tuple`	27m	1x	4.7x	1.9x

Conclusion

1. Manipulation the tree is obtained at immense effort
2. Works in the range of acceptably well (better than most alternatives) to exceptionally well (better than hand coded)
3. Prunes global tree (computation optimisation)
4. Orders data by access (cache optimisation)
5. Compile-time features of the language are too limited to use this approach neatly (cf. preprocessor macros)
6. Inlining gives up too readily; `__attribute__((always_inline))` used ubiquitously
7. Only supports having one result
8. Not obvious what is going on with `std::tuple`

Float template parameter

Distinguishing different constants



⚠ Values as types ⚠

```
// C++ does not permit 'auto' to resolve as 'float'
template<auto V>
struct _float
{
  constexpr operator auto() const { return V; }
};
```

```
// type distinguished by value
auto constexpr c0 = -4.2_f;
static_assert(c0 == _float<-4.2>{});
```

Values as types in D

```d
// exactly what is wanted
struct _float(float v)
{
  static immutable auto value = v;
}
```

"_float" workaround

```
// exponent ignored...
template<auto H, auto L, auto E>
struct _float
{
  auto constexpr static value =
    (H + float(L) / multiplier<10, E, 1>::value);

  constexpr operator auto() const { return value; }
};
```

```
// and for operator+
template<auto H, auto L, auto E>
auto constexpr operator-(_float<H, L, E>)
{
  return _float<(-H), (-L), E>{};
}
```

. . . made palatable

```
// makes life easier
template<char...> struct mp_chars {};
```

```
// user-defined literal
template<char... Cs>
auto constexpr operator""_f()
{
  return make_float_t<0, 0, 0, 0,
                      sizeof...(Cs), mp_chars<Cs...> >{};
}
```

```
// seamless conversion to literals
auto constexpr c0 = -4.2_f;
float t0 = 2 * c0;
```

Reflect variable location

The trusty preprocessor

```
auto fn(A const &a, B const &b)
{
  auto c0 = UQ(7); // auto = Unique<724, float>{7}
  auto c1 = UQ(9); // auto = Unique<725, float>{9}

  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;
  return t1 / t2;
}
```

A fundamental problem

Write a `transform` function to yield:

```
auto c0 = 7;
auto c1 = 9;

transform(c0 + c0)  -->   2 * c0
transform(c1 + c0)  -->  c1 + c0
```

Impossible! . . . despite it appearing so trivial

Three kinds of reflection?

```
 123456789 123456789 123456789
         10        20        30
1
2  // main.cpp
3
4  decltype(x)*   y   =   &x;
5

     reflect             reflect
     type                address
```

Reflect *location* of `y`, at `[18, 4, main.cpp]`?

Reflect location in D

```
struct Terminal(T, string file = __FILE__,
                   size_t line = __LINE__)
{
  T value;

  ...
}
```

```
Terminal!float c0 = 7; // Terminal!(float, "main.d", 724)
Terminal!float c1 = 9; // Terminal!(float, "main.d", 725)
```

One definition per line could be run-time checked with a debug build which requires all `Terminals` be singletons.

Compile time adjoint in C++

22nd EuroAD Workshop, Imperial College

Dominic Jones

`github.com/DominicJones`

July 2019