Expression tree transforms

For compile-time differentiation

Dominic Jones

dominic.jones@gmx.co.uk

March 2018

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

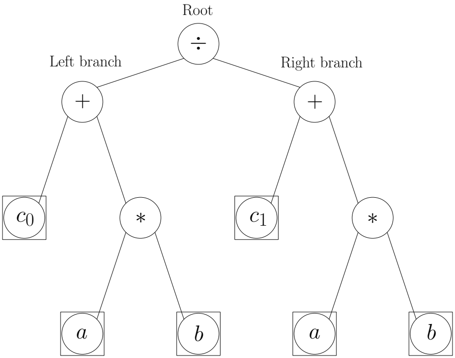
1. Given a pure-functional algorithm, differentiate it
2. Implement the transpose of the chain of derivatives (the adjoint)
3. The required ‘extra’ code is in the reversed sequence of the original and the data flow is reversed
4. Eager and lazy evaluation: ctor-dtor pairs?

Two hurdles

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; // l.b.
  auto t2 = c1 + t0; // r.b.

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



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

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

```
mul(A const &a, B const &b)
{
  auto cm = 1; // lost on return!
  return cm * a * b;
}

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

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

  r  = t1 / t2;
}
```

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 = 1;
  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 = 3;
  auto c1 = 4;
  auto cm = 1;
  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

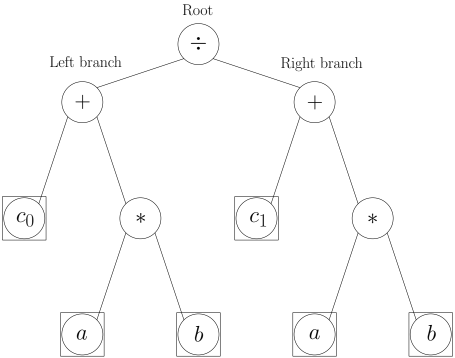
Only capture terminals? - But how to know if they are already captured?

A ‘unique types’ approach

Tagging terminals

With all terminal types uniquely tagged, any duplicate branches can be identified

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

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

  r  = t1 / t2;
}
```



Unique

Ugly, but works

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

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

Interesting, yet useless

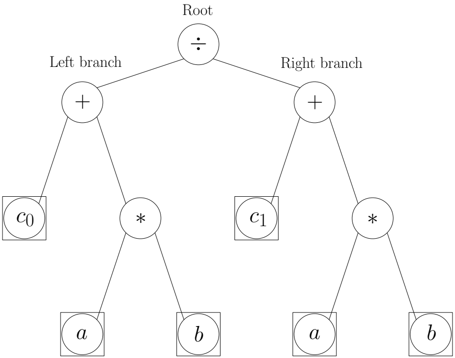
```
template<typename T, typename U>
auto constexpr
cmp(T const &t, U const &u)
{
  // Not possible!
  // auto constexpr ta = &t;
  // auto constexpr ua = &u;
  // return ta == ua;
  return &t == &u;
}

{
  auto c0 = 7;
  auto c1 = 9;

  static_assert(cmp(c0, c0));
  static_assert(!cmp(c0, c1));
}
```

Building the components

Bookkeeping

Types and data to percolate up the tree:

1. Hierarchically grouped node types (`Binary<Mul, A, B>`, etc)
2. List of constant values (`c0` and `c1`)
3. List of argument addresses (`a` and `b`)



Hierarchically grouped node types

In the root node:

```
// Pad left and right branches to equal depths
```

```
// Merge left and right branches:
  {{A, B, A, B},
   {Binary<Mul, A, B>, Binary<Mul, A, B>},
   {Binary<Add, C0, ...>, Binary<Add, C1, ...>}}
```

```
// Prune duplicates:
  {{A, B},
   {Binary<Mul, A, B>},
   {Binary<Add, C0, ...>, Binary<Add, C1, ...>}}
```

```
// Augment with self:
  {{A, B},
   {Binary<Mul, A, B>},
   {Binary<Add, C0, ...>, Binary<Add, C1, ...>},
   {Binary<Div, ...C0..., ...C1...>}}
```

Hierarchically grouped node types

1. Prune duplicate branches eagerly
2. TMP operations exclusively on types
3. Relatively low compile-time overhead
4. Moderately involved: `resize`, `unique_merge`, `append`
5. Using the Brigand library

List of constant values

In the root node:

```
// Identify the longer list:
  LLC = get_longer<L::LC, R::LC>   i.e. {C0}
  SLC = get_shorter<L::LC, R::LC>  i.e. {C1}
```

```
// Convert lists to sets:
  LSC = brigand::as_set<LLC>
  SSC = brigand::as_set<SLC>
```

```
// Merge sets and convert to tuple:
  TC = rename<merge_sets<LSC, SSC>, tuple>
  TC const tc;  i.e. tuple of constants
```

```
// Difference between tuple and longer list:
  S = drop<TC, size<LLC> >  i.e. {C1}
```

```
// Construct ‘tc’ member: (biggest source of inefficiency!)
  tc{tuple_cat(get_longer(l.tc, r.tc),
               select(S{}, get_shorter(l.tc, r.tc)))}
```

List of constant values

1. Heavy work done on smaller data
2. TMP operations on types and data
3. Significant compile-time overhead due to data operations
4. Bridging the type-value divide with `select(S, L)`
   This should be optimised

`select`

```
// a naive approach...

template<
  template<class...> class S, class... SS,
  template<class...> class L, class... LL>
S<SS...> select_impl(S<SS...>, L<LL...> l)
{
  // std::get will not work
  return S<SS...>{alt::get<SS>(l)...};
}

template<class S, class L>
S select(S s, L l)
{
  return select_impl(s, l);
}

using S = std::tuple<bool,float>;
using L = std::tuple<char,bool,int,float>;

select(S{}, L{'0', true, 2, 3.0});
```

Possible optimisation: Given that types are unique and ordered, if any `S` is found in `L`, `L` can be left-cropped

Better to avoid `std::get<T>` altogether

Doing something with all of it

Assigning to the result

```
template<typename T>
struct Result
{
  template<typename Expr>
  void operator=(Expr &expr)
  {
    // depth-grouped node list
    using LN = Expr::LN;

    // list of constants
    using LC = Expr::LC;
    auto &tc = expr.tc;

    // list of argument ptrs
    using LA = Expr::LA;
    auto &ta = expr.ta;

    // Now what?
  }
};
```

Now construct the index arrays to access data:

1. build left and right node lists
2. build offset arrays for left and right node lists
3. build offset array for constants, in order to construct its dual

Converting to offsets

```
// Flatten the node list (6 elements)
  LN =  {A, B,
         Binary<Mul, A, B>,
         Binary<Add, C0, ...>, Binary<Add, C1, ...>,
         Binary<Div, ...C0..., ...C1...>}
```

```
// Offsets of left child nodes ('6' = null marker)
  INL = {6, 6, 0, 6, 6, 3}

// Offsets of right child nodes
  INR = {6, 6, 1, 2, 2, 4}
```

```
// With the list of constants...
  LC =  {Binary<Add, C0, ...>, Binary<Add, C1, ...>}

// generate node list offsets...
  IC =  {3, 4}

// then construct its 'dual' ('2' = null marker)
  DC =  {2, 2, 2, 0, 1, 2}
```

‘Eager and lazy’ evaluation

Inside the result assignment operator (pseudocode):

```
// Initialise array/tuple data (6 elements)
  array<double, 6> v = 0;

// set input from tuple of argument ptrs
  for (i = 0; i != ta.size(); ++i)
    v[i] = *ta[i];
```

```
// evaluate operators
  for (N: LN)
    I = offset<N, LN>;
    N::evaluate<I>(v, tc, INL{}, INR{}, DC{})
```

```
// extract result from ‘root’ node
  this.r = v[v.size()-1];
```

```
  ... then the reverse order to differentiate
```

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 exceptionally well (better than hand coded) to acceptably well (better than alternatives)
3. Compile-time features of the language are too limited to use this approach neatly
4. Inlining gives up too readily: `__attribute__((always_inline))` used ubiquitously
5. Not obvious what is going on with `std::tuple`

Footnotes

Learning the ropes

1. Peter Dimov's "Simple C++ metaprogramming" articles
2. Time the compilation for large data sets
3. An optimised `reverse` function is critical
4. Brigand library is an excellent resource

Reflect compile-time literals

Possible

```
// Boost Hana extension...
#ifdef CONFIG_ENABLE_STRING_UDL


auto constexpr x =
  hana::string_c<'a','b','c'>;

auto constexpr y = "abc"_s;

CONSTANT_CHECK(x == y);
```

Desirable

```
template<auto v>
struct Literal {
auto constexpr static value = v;
};

auto constexpr x =
  Literal<4.2>;

auto constexpr y = 4.2_f;

static_assert(
  std::is_same_v<decltype(x),
                 decltype(y)>);
```

Reflect compile-time location

Possible

```
 123456789 123456789 12345
         10        20
1
2  // main.cpp
3
4  decltype(x)* y = &x;
5
   reflect          reflect
   type             address
```

Desirable

```
 123456789 123456789 12345
         10        20
1
2  // main.cpp
3
4  auto constexpr y = &x;
5
                      reflect
                      location


y = hash(row, column, filename)
  = hash(4, 23, "main.cpp")
```

Function signature pattern

```
struct _1;  struct _2;
template<template<class...> class, class...> struct bind;

// Apply a function taking two args to a list...
template<class L, class P> struct apply;

template<template<class...> class L, class T, class... Ts,
         template<class...> class F>
struct apply<L<T, Ts...>, bind<F, _1, _2> >
{
  using type = L<F<T, Ts>...>;
};

// A function taking two args...
template<class T, class U>
using cmp = bool_t<(sizeof(T) > sizeof(U))>;

// evaluate 'cmp' over a list
using L = list<int, bool, char, double>;
using R = apply<L, bind<cmp, _1, _2> >::type;
```