Compile Time Differentiation Design

Dominic Jones

March 29, 2018

1 Specification

Previous work on constructing the adjoint from a generated expression tree made use of destructors to trigger the evaluation. This approach works reasonably efficiently relative to its equivalent hand coded adjoint. The approach, however, had a fundamental limitation of not supporting nested function calls due to locally scoped variables prematurely triggering their adjoint evaluation upon destruction. Whilst the destructor approach, then, seemed to be the ideal solution to computing the adjoint, it also acted as the most significant hindrance to expanding functionality because of its implicit behaviour.

To resolve the nested function problem and to further improve the performance of the tool such that it could be on par with hand coded adjoint, a new approach was needed which did not rely on leveraging the destructors to trigger adjoint computation.

2 Design

To achieve on par performance with hand coded adjoint, the following constraints were followed:

1. work within the type system so the compiler can optimise as much as possible,
2. minimise the object size of the root node to prevent object copying dominating the execution time,
3. keep the memory layout well aligned, local, and always on the stack,
4. avoid `if` statements or conditional operators to manage evaluation,
5. aim to make the task of compiler optimisation nothing more than the micro-optimisation of inlining function calls,
6. avoid delegates or any form of call-back techniques.

3 Implementation

For the illustration of implementation details, Figure 1 and its equivalent code in Listing 2 will be considered. Suppose the function in Listing 1 was required to be differentiated. The aim of the differentiation tool is such that by providing the appropriate `Mode` type and including the header file of the tool, the primal, tangent or adjoint function would be compiled.

```
#include <exprlist.h>
template<class Mode>
void compute()
{
  Terminal<Mode, A const> a;
  Terminal<Mode, B const> b;
  Terminal<Mode, R> r;
  r = evaluate(evaluate(a, b), evaluate(b, a));
}
```

Listing 1: Nested function calls must propagate their expression trees.

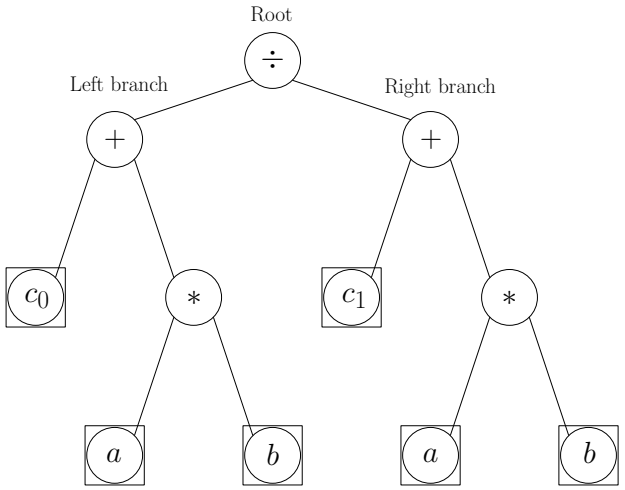

Figure 1: Duplicate branch? Impossible to tell given only the types.

```
template<class TA, class TB>
auto evaluate(TA a, TB b)
{
  auto c0 = 3;
  auto c1 = 4;
  auto ab = a * b;
  auto lb = c0 + ab;
  auto rb = c1 + ab;
  return lb / rb;
}
```

Listing 2: `lb` and `rb` have the same type but do not represent the same expression.

3.1 Building the tree

To evaluate the root in any mode, the expression tree must

1. build up the full type of the expression
2. capture the addresses of the active terminal nodes
3. capture the values of the passive terminal nodes

Typical expression template tools will store the intermediate state at each node, though this is not necessary. Intermediate state is only required when evalution of the root is triggered, and so any optional allocation can be delayed until then.

3.1.1 Expression list

Building up of the expression type is straight forward: by using `auto` for the variable and return types, each new expression nests the types of its arguments. One problem exists with this, however. The expression type is nothing more than a type, and therefore conveys no information about whether or not two or more identical types found in the full expression type are semantically the same thing. Figure 1 demonstrates the problem: how can duplicate subexpressions be identified in order to avoid duplicate evaluation? With only type information it is not possible since the values may be different. Since comparing memory addresses of subnodes of same type would violate the design requirements (causing excessive run-time overhead), the alternative was to somehow encode into the type the identity of the subexpression such that duplicate subexpression matching can be correctly performed.
Listing 3 offers a solution, though using it requires decorating passive values with the preproccessor function (i.e. `auto c0 = UQ(3);`). Furthermore, template functions are needed to evaluate mathematical operations performed on passive values decorated with the unique type and return new unique types with the resulting value.

```
// simple class to identify unique types
template<std::size_t ID, class T> struct Unique { T value; };

// hide __COUNTER__ behind macro
#define UQ(value) Unique<__COUNTER__, decltype(value)>{value}
```

Listing 3: `__COUNTER__` offers a non-standard way to distinguish variables.

Whilst building the expression type poses no difficulties, building the unique list of operations does. This task is a type transformation, converting the expression tree as it is being built. Considering Figure 1, its expression type would like that of Listing 4, whereas the expression list would be like that of Listing 5: a list of lists order according to depth, with each depth containing unique types.

```
Binary<Div,
  Binary<Add,
    Unique<0, double>,
    Binary<Mul, A, B> >
  Binary<Add,
    Unique<1, double>,
    Binary<Mul, A, B> > >
```

Listing 4: Expression type of Figure 1.

```
// template<class... Ts> struct list {};
list<
  list<A, B>,
  list<Binary<Mul, A, B> >,
  list<Binary<Add, Unique<0, double>, ...>,
       Binary<Add, Unique<1, double>, ...> >,
  list<Binary<Div, ..., ...> > >
```

Listing 5: Expression type list of Figure 1.

A problem not encountered in this simple case is where one subexpression is deeper than the other. Merging of the two sublists can only be performed if they are the same length. To remedy this problem, both sublists are first resized to the larger of the two, and padded with empty nested lists if required.

3.1.2 Active terminals

Capturing the addresses of the active terminal nodes efficiently is non-trivial as both types and data are involved. Considering again Figure 1, the terminal nodes are repeated in the left and right branches. This duplication needs to be avoided and so at every node which is not a unary operation, a merge operation of the addresses must be performed. Ordinarily, the number of unique active terminals will be small regardless of the size of the expression tree, so little benefit is gained from having a highly efficient implementation of the merge operation.

3.1.3 Passive terminals

Capturing the values of the passive terminal nodes is essentially the same task as capturing the addresses of the active terminals. The only significant difference here is that the efficiency of the merge operation does become a factor as the number of values typically scales with the size of the tree. For both this and the merging of the active terminals the same function is used. In Listing 6 two sets are merged: `s1` and `s2`. Since at least all of one of the sets will be included in the new set, the bigger of the sets is selected immediately. The next task is to find any of the members of the smaller set in the larger set. If there are any that are not found, add them to the new set. This search operation is presently implemented naively: each memeber of `s2` is searched for in `s1`. However, this could be improved: given that elements are always in the same order (in this case `char` is always followed by `bool`) and elements are unique, the search space in `s2` can be reduced after each elemental search of `s1`. So, for `char` in `s1`, all elements in `s2` must be searched, but for `bool` in `s1` only the third and fourth elements in `s2` must be searched, and finally, for `double` in `s1`, no search needs to be performed since the `bool` found in the previous search was already at the upper bound of `s2`.

```
std::tuple<char, bool, double> s1{'c', true, 5.6};
std::tuple<int, char, float, bool> s2{2, 'c', 3.4, true};

// std::tuple<char, bool, double, int, float>
auto s3 = merge(s1, s2);
```

Listing 6: A simplified example of merging two sets.

3.2 Evaluating the list

Building the expression list accounts for much less than half of the compiler work; the bulk of the compilation is building the index lists for the intermediate data and passive values and secondly for looping over the node operations in order for the primal then in reverse order for the adjoint.

3.2.1 Indexing the expression list

For unary, binary and ternary nodes, their left, middle and right data indices are required. Given the list from Listing 5, the index lists would be:

```
// from:
list<A, B, Binary<Mul, ...>, Binary<Add, Unique<0, double>, ...>, Binary<Add, Unique<1, double>,
    ...>, Binary<Div, ...> >

// construct: (ic -> std::integral_constant)
list<ic<6>, ic<6>, ic<0>, ic<6>, ic<6>, ic<3> > // left indices
list<ic<6>, ic<6>, ic<1>, ic<2>, ic<2>, ic<4> > // middle indices
list<ic<6>, ic<6>, ic<6>, ic<6>, ic<6>, ic<6> > // right indices
```

Listing 7: Generating the left, middle and right indices.

Since `A` and `B` have no subexpressions, the left, middle and right indices are always set to the out of range value (i.e. the size of the expression list after flattening it), and likewise for the passive terminals. Also, since there are no ternary expressions in the example then all the right indices are out of range. Implementing the function to generate these index lists is fairly involved though efficient: for any given node its subnodes can be deduced and from them their depth. Since the expression list is depth ordered then only the elements in the specific depth need to be searched.

3.2.2 Indexing the passive values

Indexing the passive value data follows the same first step as indexing the subexpressions, but there is an additional step that must also be performed. Again, using the same example, the required index list would be:

```
// from:
list<A, B, Binary<Mul, ...>, Binary<Add, Unique<0, double>, ...>, Binary<Add, Unique<1, double>,
    ...>, Binary<Div, ...> >

// construct:
list<ic<3>, ic<4> > // indices (step 1)
list<ic<2>, ic<2>, ic<2>, ic<0>, ic<1>, ic<2> > // dual of 'indices' (step 2)
```

Listing 8: Generating the passive value mapping.

3.2.3 Evaluating the operations

Evaluation of the full expression list is performed by recursively stepping through each node in the list, first in order to compute the primal, and then in reverse order to compute the adjoint. Intermedate data both for the primal and adjoint is stored in `std::tuple`s, so accessing elemental values requires using `std::get`. This has turned out to be very expensive to compile for large expression trees, compared to if the data was simply stored in `std::array`s and the subscript operator used to access values.

One possible improvement to minimise the compilation time due to the use `std::get` for very large lists is to evaluate the full expression list as a list of lists ordered by depth. This would mean that `std::get` is operating over a much smaller range and so should reduce compile time despite double indirection being required.

4 Further work

4.0.1 Multiple assignments

The present method only efficiently supports one assignment because it is at the point of assignment that all the computation is performed therefore having more than one assignment will almost inevitably mean duplicate evaluations being performed. There is no obvious work-around to this problem. Whatever the solution though, somehow it must bring together the different output terms so as to make them visible with a single assignment operation. Furthermore, probably some sort of checkpointing on the roots of the results would be required.

Finally, though it would be very contrived, some way of bringing the various roots into a single root and the evaluating like normal on that single root would work. This might mean adding a special operator which can work as a mechanism to seed the result roots and also join it to the the final root.

```
evaluate2(A a, B b, R1 &r1, R2 &r2)
{
  auto t1 = a + b;
  auto t2 = (a * b) / t1;

  // cache adjoint result seeding values
  auto expr_r1 = assign_result(r1, t1);
  auto expr_r2 = assign_result(r2, t2);

  // combine subexpressions (using addition)
  // and evaluate
  evaluate_tree(expr_r1, expr_r2);
}
```

Listing 9: Combining multiple roots into one.

4.0.2 `std::get` alternative

The standard library implementation for accessing elemental data in a `std::tuple` incurs a relatively long compile time and performs relatively poorly to a homogeneous container, such as `std::array`, even when the types in the tuple are identical. Efforts to remedy the poor compilation time have been tried by implementing an alternative hetrogeneous container, along with accessor functions. This alternative implementation remedied the compilation time, effectively halving it for large tests, but inlined poorly, resulting in relatively slow run times.

Boost Hana offers its own tuple implementation which claims to have both good compile time and run time performance, though it has not been tested in this work.

4.0.3 Merge operation

The merging of the passive values for a binary node presently incurs the single biggest run time performance loss. This is highlighted by comparing the run time of a large case with the using the present constructor and with the default constructor. Clearly, in the second case the results will be meaningless, but the test highlights that the constructor operation is not getting inlined. The reason for this is likely to be due to the amount of compilation work required to

implement the constructor. As noted above, there are improvements that could be made to trim down the amount of unrolling required for the merge operation. This task, though difficult, would probably yield the single biggest benefit to the methodology.

4.0.4 Recursive evaluation

The operation list is iterated naively using recursion for both primal and adjoint evaluation, and its compilation accounts for a major proportion of the overall time. Other techniques exist for iterating over lists, such as using multiple inheritance or the function signature matching technique. These may yield meaningful compilation time improvements, though none has been tested.