Compile Time Differentiation

Dominic Jones*

August 7, 2017

Abstract

To ease the burden of manually differentiating arithmetic expressions, typically required for implementing adjoint solvers, a methodology is presented which automatically performs the differentiation of an expression or a block of expressions and yields more efficient machine code than its equivalent source transformation implementation [4]. The methodology leverages template metaprogramming techniques to provide a means of generating the differentiated statements whilst facilitating near perfect inlining of code. Whilst the the functionality of the methodology is limited, it presents what level of run-time performance is achievable.

Keywords: Expression Templates, Operator Overloading, Automatic Differentiation, Adjoint, C++, Template Metaprogramming

1 Introduction

Two dominant methodologies exist for computing the differential of an algorithm without having to explicitly program its derivative: the first and more popular is to replace the floating point type of the algorithm with another type which augments the relational behaviour of the original floating point type with the side effects of recording every operation and associated state to some unique stack [3]. The derivative is evaluated by interpreting the stack after the algorithm has completed its execution. This method is commonly referred to, in context, as operator overloading. The second methodology is to use a tool to read the source code which implements the algorithm, and have the tool generate new source code which contains the implementation of the original algorithm and its derivative [4].

Both methods have significant limitations with respect to their application to large industrial codes: the first method destroys compiler optimisation opportunities due to its side effects and its exclusively run-time governed interpretation of the heap-stored stack. The second potentially retains the high-performance characteristics associated to manually implemented differentiated code, but is generally difficult, if not impossible, to apply it to codes written modern system programming languages due to their expansive complexity.

Performance is a perennial issue in industrial numerical software, where enormous computational resources are dedicated to solving simulations. It would not be unreasonable for the user of such software to expect the adjoint of a simulation to take about the same time as the primal simulation itself and requiring a similar memory footprint. To achieve this parity, both the primal algorithm and its adjoint would need to have similar implementation characteristics and compile to highly efficient machine code. A compromise on efficiency from the outset regarding how the adjoint is implemented would hinder the adoption of the feature, especially for industrial users to routinely simulate problems close to the maximum capacity afforded by their systems.

In addition to these difficulties, there is the often overlooked but critical issue of program build varieties. It is already common to have two versions of a numerical analysis software, one compiled with mixed precision floating point types and the other with double precision types. With a continuous delivery build system, every variety of the released program must be compiled and tested on a nightly basis. If another version was added, such as build using a differentiable double precision type, the resources required to absorb this extra work load may very well outweigh the perceived benefit in the added functionality being delivered.

With these constraints in mind, hand coding the required differentiated components appears to emerge as the path of least resistance for obtaining the desired adjoint solver. It is under this perspective that the following work pursued, whose aim is of offering some helping-hand, in the context of a C++ codebase, to carrying out the task whilst minimising performance compromises.

This is not the first attempt to produce a high-performance adjoint differentiation library for C++ [1], however this paper seeks to document one approach to implementing a solution.

2 Methodology

Consider the contrived function in Listing 1 which has two input variables and one output variable. The proposed methodology enables the function to compute the primal and its adjoint derivative by simply calling the function; no recording of operations is performed at run-time.

*Netherhall House, London NW3 5SA, dominic.jones@gmx.co.uk

```
1  void example(Terminal<A> const &a,
2               Terminal<B> const &b,
3               Terminal<R> &r)
4  {
5    // unique types for passive values
6    auto const c0 = UQ(5);
7    auto const c1 = UQ(6);
8
9    // generate expression tree
10   auto const tmp0 = c0 * a * sin(a) / cos(b) * c1;
11   auto const tmp1 = arg1 * c0 * sin(b) / UQ(7) * cos(a);
12   auto const tmp2 = sin(tmp0) * tmp1 + sin(tmp1) * tmp0 * c1;
13   auto const tmp3 = sin(tmp2) / cos(tmp2)  + cos(tmp0) / sin(tmp0);
14
15   // evaluate upon assignment
16   r = tmp3;
17 }
```

Listing 1: All terminal nodes require unique types, both for active and passive values.

2.1 Destructors Approach

The order of destructor calls of the *l-value*s (i.e. `tmp0`, etc) is exactly the order required for adjoint computation. However, leveraging this feature of C++ for this purpose is deliberately avoided in this methodology. The reason for this decision is that it is impossible to prevent double evaluation of the destructor in the case of a term existing as a temporary variable. The first destructor evaluation occurs when the temporary variable goes out of scope at the end of a statement. Here, no adjoint computation is wanted as it is premature. The second is when its copy goes out of scope at the end of the block scope, which is when the adjoint computation is wanted. Preventing one but not the other, however, would require a run-time flag, effectively ruining the sought high-performance wanted in this work.

If, on the other hand, named types are used instead of `auto`, it is possible to avoid the run-time flag by making use of type slicing, offering good performance, but still yields sub-optimal performance [6].

2.2 Location of State

Typical implementations of expression trees hold the intermediate state of the unary or binary operations [5]. However, they need not do so. The absolute minimum of state is simply the memory addresses of the terminals and the values of the passive terms. This is the approach used here, and the reason for taking this approach is to maximise the likelihood of all the copy constructors being inlined when the tree is being built. If the nodes hold state then as the tree gets larger the copying operation becomes on par with the computation. One way to assure such excessive work is avoided is to not hold intermediate state in the first place. Storage of intermediate values will be required at some point, but the (static) allocation is delayed until the result assignment is reached.

2.3 Leveraging Tree Structure

A naive implementation of expression tree evaluation would implement a *compute all* strategy, i.e. all branches, whether or not they are duplicate branches, would be evaluated. Such duplicate branches would occur when a named variable is used more than once, or an intermediate (temporary) expression occurs more than once. The above mentioned *destructor approach* circumvents this duplicate evaluation because intermediate primal values are computed eagerly and stored locally during construction and the adjoint is evaluated exactly once during destruction of each *l-value* node.

This approach, storing no intermediate state and avoiding the use of destructors, trims duplicate branches by recording the node types in a unique type list as the tree is being constructed. If all disctinct terminals in the tree have unique types then duplicate branches can be safely pruned, thus providing at the root a list of exactly the nodes which need evaluating.

Building this list of unique types, however, requires attention. The order of adjoint evaluation is critical, and it is not simply the naive reverse-order of the list of node types built during construction. Rather, the list must be ordered by node *depth*. This important detail introduces a significant complication into the implementation, but at the same time is the means by which sensible compilation times can be achieved.

Consider the `Binary` node class in Listing 2. Two approaches could be used to build the `node_list_t` type: a flat, one dimensional list of types ordered by depth, or a two dimensional list of lists, where each subgroup contains types of common depth. The latter approach is used here in order to facilitate search operations on the list when tree is finally ready for evaluation. (`mp_list` is an empty variadic template class.)

```
1  template<typename Fn, typename L, typename R>
2  struct Binary
3    // Node depth accessed via Binary::value
4    : std::integral_constant<int, std::max(L::value, R::value) + 1>
```

```
5  {
6    // Increase the child node lists to make them of equal length
7    using _l = mp_resize<Binary::value, typename L::node_list_t, mp_list<> >;
8    using _r = mp_resize<Binary::value, typename R::node_list_t, mp_list<> >;
9
10   // Merge the node lists, erasing any duplicates
11   using _lr = mp_uniq_merge<_l, _r>;
12
13   // Append this node (as a subgroup) to the end of the list
14   using node_list_t = mp_append<_lr, mp_list<mp_list<Binary> > >;
15
16   ...
17 };
```

Listing 2: Template metafunctions are used extensively to build the node list.

For nodes whose subnodes are both active, no additional state is introduced. However, in the case of a binary operator taking a passive value as an operand (Listing **??**, this value must be stored somewhere. Storing the value on the node itself is a possibility but is avoided so as not to incur the cost of having to store pointers to all nodes. Passive values are, instead, accumulated in a stateful list (`std::tuple`). Having this arrangement introduces the requirement of mapping from the *node_list* to the passive value list, which are not necessarily in the same order, since the former is depth ordered and the latter is constructor invocation ordered.

```
1  template<typename Fn, typename L, std::size_t ID>
2  struct Binary<Fn, L, Unique<ID, double> >
3    : std::integral_constant<int, L::value + 1>
4  {
5    // augment the list type of passive values
6    using passive_list_t = mp_append<typename L::passive_list_t,
7                                     std::tuple<Passive<Binary, double> > >;
8
9    // concatenate and store augmented list
10   Binary(L const &l, Unique<ID, double> const &r)
11   : passive_list(std::tuple_cat(
12       l.passive_list, std::make_tuple(Passive<Binary, double>{r.value})))
13   {}
14
15   passive_list_t const passive_list;
16
17   ...
18 };
```

Listing 3: Mixed arithmetic of active and passive operands.

2.4 Unique Types

Unique types for every terminal, including passive value terminals, are required because when the *node_list* is being constructed, duplicate nodes are removed. The assumption in removing duplicates is that they unambiguously correspond to certain numerical terms, which can only be guaranteed if every terminal is uniquely tagged. The following contrived example (Listing 4) highlights the issue.

```
1  void example(Terminal<A> const &a,
2               Terminal<B> const &b,
3               Terminal<R> &r)
4  {
5    auto const tmp0 = a * b;
6    auto const tmp1 = tmp0 + 3.0d;
7    auto const tmp2 = tmp0 + 4.0d;
8    r = tmp1 / tmp2;
9  }
```

Listing 4: Names cannot be extracted and so the type must be the carrier of identity.

The types of `tmp1` and `tmp2` are identical (i.e. `Binary<Add, Binary<Mul, Terminal<A>, Terminal<B>>, double>`), however, numerically they are distinct. Since the numerical distinction cannot be communicated at compile-time to the nodes being constructed, then without further measures, the result `r` would simply become `tmp1 / tmp1` (assuming that `tmp2` is the erased duplicate).

The measure taken here to prevent this problem is to require that passive values be first wrapped in a unique type, (i.e. `Unique<int ID, typename T>`). This ensures that different passive values are treated differently, though it does mean that identical temporary values are treated differently, too, leading to a (probably negligible) source of inefficiency.

The `ID` template argument of `Unique` is defined by `__COUNTER__`, which is a non-standard preprocessor macro (though is supported by GCC, Clang and MSVS). All of this is in turn wrapped up the macro function definition of `UQ`. The counter is incremented on every use, providing a sure mechanism to generate the unique types for the passive values.

The value itself is of no consequence and so could be any unique number. Standard preprocessor macros that could partially emulate the role of `__COUNTER__` are `__LINE__` and `__TIME__`. However, whilst the the line number effectively would suffice, it would not guarantee correctness. The second would suffice but for the the fact the it only resolves to seconds (and is a formatted string).

Having to wrap passive values in a non-standard preprocessor macro is the weakest aspect of the design, forcing the programmer to use a syntax that he would not expect to be needed. No obvious work-around is known, despite the fact that it is statically unambiguous where a named variable is being used.

Whilst it is possible to inject unique identity (unary) nodes ahead of every node holding a passive value on the tree (using template expression tree transform methods) without having to make use of preprocessor macros and unique type wrapper classes, the the functionality is still unsatisfactory. This approach has the effect of requiring the the recording of every occurence of every passive value that appears on the *node_list*, which quickly becomes overwhelming for large trees.

Given that within the functionality that C++ presently offers (2011, 2014, 2017 standards) a viable solution cannot be implemented to solve the above described problem, a small language extension is proposed. In Listing 5, if a new operator was defined, call it `varid()`, which returned the compiler's internal index of the variable given it, then all unique terminals could be identified *and* identical terminals could be identified, too.

```
1  template<typename Fn, typename L, typename R, uint64_t IL, uint64_t IR>
2  struct UniqueBinary
3  {
4    UniqueBinary(L const &l, R const &r);
5  };
6
7  template<typename L, typename R>
8  auto operator*(L const &l, R const &r)
9  -> UniqueBinary<Mul, L, R, varid(l), varid(r)>
10 {
11   return {l, r};
12 }
```

Listing 5: A possible language extension of `varid()`, returning the internal index for a given variable.

2.5 Evaluating the Tree

Since neither the primal is evaluated upon construction of the nodes, nor the adjoint upon destruction of the *l-values*, all evaluation must be performed upon assignment to the result variable. This arrangement has a significant advantage and at the same time a significant disadvantage. The advantage is that the methodology works seamlessly with nested function calls (using `auto` return type in C++14). This is facilitated because local variables in nested functions are not referenced by the tree, so when those variables go out of scope there is no risk of a segmentation fault. The disadvantage is that only a single assignment can be cleanly and efficiently supported. If more than one assignment is used then the tree will be evaluated again (from that required root node).

Within the assignment operator of the result variable, primal and adjoint state are allocated on the stack, child indexing of the *node_list* children are computed, and the mapping of the *node_list* passive binary node types to the *passive_value* list is determined. Once these steps are complete, the *node_list* is iterated in order to compute the primal result, and then in reverse order to compute the adjoint derivatives.

2.6 Implementation Details

Adjoint specific code in this methodology is both a negligible amount and is trivial to implement. The vast majority of the code is template metafunctions for performing primary and secondary level operations on variadic template types. Most primary functions can be found in modern template metaprogramming libraries. In this work Brigand [2] has been used due to its flexibility. Secondary level algorithms were written specifically for this work, such as two dimensional searching, unique merging, graph dual construction, and list resizing.

3 Performance

To test the methodology outlined in this paper, Listing 6 was used as a prototype function. To compare the run-time performance of the adjoint, the source code (after lowering it to C code) was supplied to Tapenade [4], a source transformation differentiation tool. Table 1 presents the execution times of the adjoint normalised by the primal along with the primal execuation time and compilation time of the adjoint function using the compile time methodology.

The tests were compiled with both g++ 6.2.0 and clang 3.8.0 (with the `-O3` flag) on a machine running on an Intel Core i3-4130 processor. Results presented are obtained by computing the median-average run-time of 200 sets of evaluations, where each set performs 100,000 invocations of the tested function.

As is seen from the results, the timings for this approach are noticably faster than the source transformation code produced by Tapenade. A possible explination for this is that the compile time approach orders the execution of

	Primal	Tapenade AD	CT differentiation	Compilation time
g++	64.25ms	1.484	1.257	2698ms
clang	138.0ms	2.626	1.032	9108ms

Table 1: Clang compiler performs relatively poorly, though a common tread is found.

operations in increasing depth, facilitating the pipelining of independent calculations. Without such reordering, the evaluation of terms will (unless specifically optimised by the compiler) proceed in the order dictated by the rules of operator precedence, causing a measurable increase in computational time due to cache stalling.

```
1  void test(Terminal<A> const &a,
2            Terminal<B> const &b,
3            Terminal<R> &r)
4  {
5    auto const c1 = UQ(1);
6    auto const c2 = UQ(2);
7    auto const c3 = UQ(3);
8    auto const c4 = UQ(4);
9    auto const c5 = UQ(5);
10   auto const c6 = UQ(6);
11   auto const c7 = UQ(7);
12   auto const c8 = UQ(8);
13   auto const c9 = UQ(9);
14   auto const c10 = UQ(10);
15   auto const c11 = UQ(11);
16   auto const c12 = UQ(12);
17
18   auto const tmp0 = c1 * a * sin(a) / cos(b) * c2;
19   auto const tmp1 = b * c3 * sin(b) / cos(a);
20   auto const tmp2 = sin(tmp0) * tmp1 + sin(tmp1) * tmp0;
21   auto const tmp3 = sin(tmp2) / cos(tmp2) + cos(tmp0) * c4 / sin(tmp0);
22   auto const tmp4 = sin(tmp2) / c5 * cos(tmp2) + cos(tmp1) / sin(tmp3);
23   auto const tmp5 = sin(tmp2) * cos(tmp4) + cos(tmp0) * sin(tmp1);
24   auto const tmp6 = c6 * sin(tmp2) / cos(tmp5) + cos(tmp0) / sin(tmp3);
25   auto const tmp7 = sin(tmp4) + cos(tmp6) + cos(tmp0) * c7 / sin(tmp5);
26   auto const tmp8 = sin(tmp1) / cos(tmp5) * cos(tmp0) * sin(tmp6);
27   auto const tmp9 = sin(tmp6) + c8 * cos(tmp4) + cos(tmp0) / sin(tmp1);
28   auto const tmp10 = sin(tmp5) / cos(tmp3) + cos(tmp7) + sin(tmp2);
29   auto const tmp11 = sin(tmp8) * c9 * cos(tmp2) + cos(tmp0) / sin(tmp3);
30   auto const tmp12 = c10 * sin(tmp10) / cos(tmp1) + cos(tmp0) / sin(tmp4);
31   auto const tmp13 = sin(tmp12) + cos(tmp0) * c11 + cos(tmp0) / sin(tmp3);
32   auto const tmp14 = cos(tmp13 / sin(tmp0) + tmp6 / sin(tmp11) * c12 * tmp13 *
         tmp11 * tmp9);
33   r = tmp14;
34 }
```

Listing 6: The test function, making much use of transcendental functions

4 Conclusion

A tool has been developed for constructing the adjoint derivative of a function while faciliating the compilation of near optimal object code. Its functionality is limited, but what is does support, namely blocks of pure-functional numerical expressions, it supports very well.

A better way of handling passive values is wanted, though seeking a more favourable solution may require a significantly different methodology to the one presented here. Furthermore, the limitation on single assignment is stringent, so some way of being able to check-point multiple assignments in order to avoid duplicate full tree evaluations would make this approach applicable to far more scenarios.

References

[1] J. du Toit, J. Lotz, and V. Mosenkis. From Runtime to Compile Time Adjoints. Numerical Algorithms Group and RWTH Aachen, April 2015.

[2] J. Falcou and E. Alligand. Brigand C++ metaprogramming library. Compile time C++11 algorithms, June 2017.

[3] Andreas Griewank, David Juedes, H. Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[4] L. Hascoët and V. Pascual. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39(3), 2013. URL `http://dx.doi.org/10.1145/2450153.2450158`.

[5] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):26:1–26:24, jun 2014. URL `http://doi.acm.org/10.1145/2560359`.

[6] D. P. Jones. Block scope differentiation. In *7th International Conference on Algorithmic Differentiation*, Oxford, UK, September 2016. SIAM.