C++ SG7 – Reflection
`varid` operator – R1

Dominic Jones[*]

January 28, 2021

1 Introduction

Expression tree transformations at compile-time are based exclusively on knowledge of expression node types. However, to successfully perform at least one kind of transformation, the pruning of duplicate sub-expression branches, type information, alone, is insufficient. A candidate duplicate branch must be inspected to determine whether or not it is a duplicate with respect to both its type *and* its associated variables.

To the best of the author's knowledge, the latter requirement, inspecting for duplication with respect to associated variables, is impossible to achieve when at least primitive types are used in the expression. To remedy this, a language extension is herein proposed to facilitate the discrimination between different variables of the same type. The proposed solution is a new operator, `varid`, which is a constant expression of type `std::size_t`, and its value is unique to the definition of its argument. The complementing variadic form is also implicitly proposed, following a similar syntax to `sizeof`.

2 Motivation and Scope

Unique types for every terminal node, including what would otherwise be terminals of primitive type (such as `float`), are required when compile-time duplicate sub-expression elimination is performed.

Consider the simple expression tree depicted in Figure 1, as a representation of the object returned by the code in Listing 1. If terminal node a is of type A, b is of type B and both c_0 and c_1 are of type C, a sub-expression duplication rule, exclusively based on types, would identify the sub-expression $c_0 + ab$ as being identical to $c_1 + ab$. This, however, would be mathematically incorrect.

Such duplicate expression identification seldom needs performing, since expressions are typically evaluated from the terminals (i.e. a, b, c_0, c_1) to the the root; when a term is repeatedly used, such as ab, the evaluation of that same term is not repeated because it would have been eagerly computed earlier on. This is not so when back-propagation is performed on an expression (i.e. evaluating from the root to the terminals), such as in the case of the adjoint automatic differentiation methodology. From this view, the ab expression path would be visited twice, becoming a source of inefficiency.

```
template<class A, class B>
auto evaluate(A &&a, B &&b)
{
  float c0 = ...
  float c1 = ...

  auto ab    = a * b;
  auto left  = c0 + ab;
  auto right = c1 + ab;
  auto root  = left / right;

  return root;
}
```

Listing 1: Duplicate `ab` could be purged at `root`.

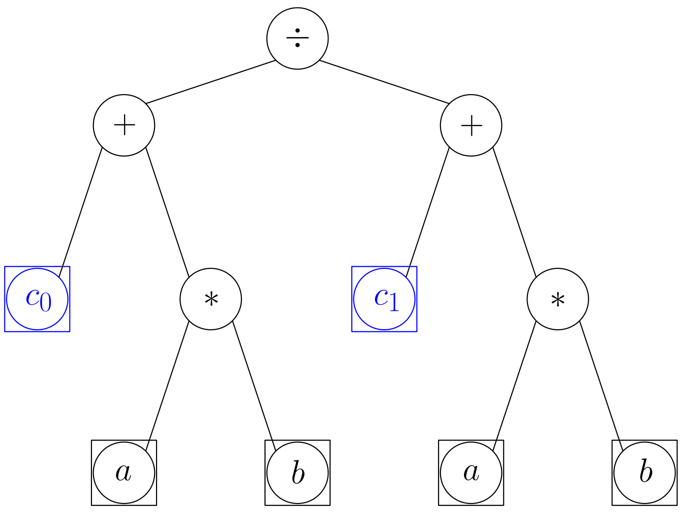

Figure 1: The mechanism for purging `ab` would somehow need to avoid purging `right`, since it is not identical to `left`.

[*]Siemens PLM, 200 Shepherds Bush Rd, London W6 7NL — `dominic.jones@cd-adapco.com`

3 Mitigating the problem

Generating unique types from lambdas reduces the need for introducing the proposed language feature. However, it does not expunge it. Furthermore, it comes with its own subtlety.

This approach requires the original primitive types (e.g. `float`) to be wrapped by template class whose second template argument is deduced from a minimal lambda definition. Since lambdas create new types, there is a guarantee that each time the wrapper *type* is fully defined, a unique type will be generated. However, the subtlety is that multiple instances of this type on the same line will all have the same type.

```
template<typename T, auto = []{}> struct Unique { ... };

Unique<float> c0;
Unique<float> c1;
static_assert(!std::is_same_v<decltype(c0), decltype(c1)>);
```

Listing 2: Creating unique types using lambdas

A second approach initially seemed promising but turned out to be of no help. The 'address of' operator can be used in a constant expression context, but only for comparison. If it were possible to leverage this feature of the language in the desired context it would have been minimally useful. Furthermore, if, in addition, it were possible to capture the values being compared it would be sufficiently useful. However, neither are possible.

The following 'address' comparison is permissible:

```
template<typename E0, typename E1>
auto constexpr isSame(E0 &&e0, E1 &&e1) { return &e0 == &e1; }

float c0;
float c1;
static_assert(isSame(c0, c0));
static_assert(!isSame(c0, c1));
```

Listing 3: Using the 'address of' operator

For it to be minimally useful in the context of determining duplicate sub-expressions, there would need to be the capacity to inject the result of `isSame` into a binary expression node type. This step is impossible since `isSame` cannot be evaluated to resolve the Boolean template parameter in Listing 4.

```
template<typename OP, typename E0, typename E1, bool isSame_>
struct Binary { ... };

template<typename E0, typename E1>
auto add(E0 &&e0, E1 &&e1)
 -> Binary<Add, E0, E1, isSame(e0, e1)>
{ ... }

float c0;
float c1;
add(c0, c1);  // error: non-type template argument is not a constant expression
```

Listing 4: Attempting to leverage 'address of'

Even if it were possible to evaluate `isSame` to resolve the template parameter, it would be of no use for the particular problem presented in Figure 1, and for the vast majority of expressions typically encountered. Suppose an expression was of the form `c0*c1 + c0*c2`. The most that could be deduced is that `c0*c1` and `c0*c2` both have different operands, but it does not tell us if their difference is the same kind of difference. Consequently, `c0*c1 + c0*c1` would be viewed in the same way as `c0*c1 + c0*c2`.

A final possibility is leveraging multiple virtual inheritance (Listing 5). The appeal of the approach is that indirectly inherited duplicate base classes, by definition, are pruned out, which is exactly what is wanted. However, one problem with using the approach occurs when directly inherited base classes are the same type, which is what would happen in the case of the example presented in Figure 1: the root node would try to inherit `Binary<Add, float, [...]>` twice. This problem is probably surmountable, perhaps by taking an approach like that used for implementing `std::tuple`, pruning duplicate types in the process. This has not yet been explored. But, supposing a solution could be found to the duplicate base class inheritance problem, multiple virtual inheritance comes with other constrictions that make the approach generally unfavourable, like having to define member data statically and, consequently, initialise them outside the class.

```
// if E0 == E1 then a duplicate base type error is triggered
template<typename OP, typename E0, typename E1>
struct Binary : virtual E0, virtual E1
{
  Binary(E0 &&e0, E1 &&e1) : E0(e0), E1(e1) { ... }
  ...
};
```

Listing 5: Attempting to leverage virtual inheritance

4 Proposal

To provide the programmer with a means to distinguish branches that have the same type but do not represent the same expression, a small language extension is proposed to resolve this problem. In Listing 6, if a new operator was defined, `varid`, which returned a value which is unique to the variable given it then variable-related distinctions of sub-expressions can be captured.

```
template<typename OP, typename E0, typename E1, auto ID0, auto ID1>
struct Binary { ... };

template<typename E0, typename E1>
auto add(E0 &&e0, E1 &&e1)
 -> Binary<Add, E0, E1, varid(e0), varid(e1)>
{ ... }
```

Listing 6: Using `varid` in the construction of a template expression node

Applying the change to the example in Figure 1, the left and right sub-expressions would have the types

```
Binary<Add, C, Binary<Mul, A, B, 1, 2>, 3, 5>  // c0 + a*b
Binary<Add, C, Binary<Mul, A, B, 1, 2>, 4, 5>  // c1 + a*b
```

Actual numbers returned by `varid` may be compiler dependent; the crucial point is that the generated expression types are different.

5 Technical Specification

`varid` takes one argument which must be a named variable or reference and returns the compiler-specific index of the variable, of type `std::size_t`. The operator name would introduce a new keyword into the language. Listing 7 presents valid use cases. In particular, the capacity to 'trace though' from a reference to a definition is key requirement. If such functionality were omitted, there would be no way to write expression node construction functions, like the one presented in Listing 6.

```
float c0;
float c1;
static_assert(varid(c0) != varid(c1));

auto &cr = c0;
static_assert(varid(cr) == varid(c0));
```

Listing 7: Valid uses of `varid`

Listing 8 presents invalid use cases. Attempting to directly use `varid` to return an index for literals or expressions is beyond the scope of the proposal. `varid` would not be supported across compilation units. If the function is used for a variable whose definition is not visible then a compilation error should be issued.

```
auto id0 = varid(float);     // error: types not supported
auto id1 = varid(3);         // error: literals not supported
auto id2 = varid(c0 * c1);   // error: expressions not supported
```

Listing 8: Invalid uses of `varid`

6 Possible Implementation

If it is the case that compilers typically have access to the line and starting column of identifiers, which appears to be so based on the types generated by lambdas, then a possible implementation would define `varid` as the combination of these two values. If, on the other hand, a compiler would need to retain this information just to speculatively support `varid` then this may be an overly expensive requirement.

```
std::size_t varid(arg)
{
  auto offset = std::pow(2, CHAR_BIT * (sizeof(std::size_t) / 2));
  std::size_t line = $arg.line;   // or however this data is accessed...
  std::size_t col  = $arg.column;
  return line + col * offset;
}
```

Listing 9: A possible definition of `varid`

The proposed implementation returns a `std::size_t` value which contains both the line and column values of the definition of the variable in the translation unit. Having the line number in the lower range of the result would preferable for analysis or debugging since it is expected that this would be the more pertinent value. Access to it could be performed by casting to the next smaller unsigned integer.

An alternative implementation may be available from Andrew Sutton's work described in P2237R0, 'Metaprogramming', p. 15. This paper describes a function, `meta::location_of(expr)`, returning line and column values. It is unclear at the moment how much overlap there is between the specification of `meta::location_of` and `varid`. However, first impressions indicate there is much.

7 Technical Issues

In the case of temporary variables being passed indirectly to `varid`, such as the `getID` function presented in Listing 10, it would be preferable for the value of `varid` to be related to the temporary that was instantiated. Otherwise, `varid` should return zero rather than triggering a compilation error.

```
template<typename E0> auto getID(E0 &&e0) { return varid(e0); }

auto id = getID(3);   // error?
```

Listing 10: Permit `varid` to be indirectly bound to a temporary?

8 Other Interested Parties

1. Marco Foco, Vassil Vassilev, et al; authors of P2072: Differentiable programming for C++.
2. Klaus Iglberger, author of Blaze — a high performance C++ math library.
3. K. Leppkes, J. Lotz, U. Naumann, and J. du Toit, authors of dco/map — a template metaprogramming library in C++ to generate adjoint code by overloading at compile time.