C++ SG7 - Reflection
`varid` operator

Dominic Jones*

August 11, 2017

1 Introduction

Expression tree transformations at compile-time is based exclusively on knowledge of node types. One operation, the pruning of duplicate branches, must assume that the duplicate branch is a duplicate as regards to its type *and also* as regards to its actual expression. The latter is impossible to achieve cleanly without some language extension to facilitate the discrimination between different named variables of the same type used as terms in a common expression. This proposal presents a possible solution by introducing a new operator, `varid`, a function used to resolve variable identity.

2 Motivation and Scope

Unique types for every terminal node, including primitive value terminals, are required when compile-time branch comparisons need to be performed. One such reason for comparing branches is to remove duplicate branches in order to avoid unnecessary repeated computation at run-time.

Consider the simple expression tree depicted in Figure 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, can an equality test be correctly performed when comparing the left branch to the right branch? Presently, the answer is *no*.

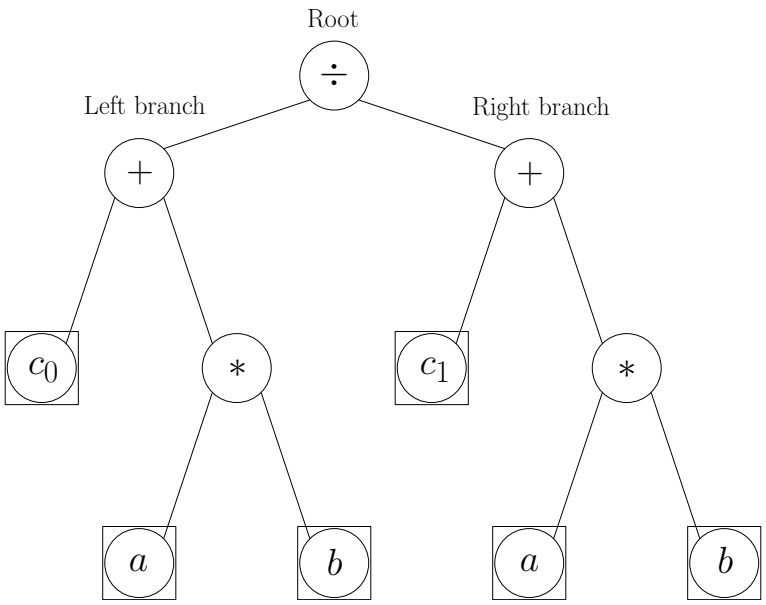

Figure 1: Duplicate branch? Impossible to tell given only the types.

At compile-time, only types can be compared for equality and not memory addresses, so a problem arises when some of the terminals in the tree have a common type but in fact refer to different variables. Taking the example of Figure 1 and assessing it in a C++ context (Listing 1), where arithmetic operators are overloaded for user-defined types to form expression types, the function would construct identical types for `t1` and `t2`, namely `Binary<Add, double, Binary<Mul, A, B> >`. If at a later stage a duplicate branch pruning transformation was performed on the returned root of the tree, the effect would be that the root would then represent `t1 / t1`, which would be numerically incorrect.

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

```
auto evaluate(A a, B b)
{
  double c0 = 3;
  double c1 = 4;

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

  return t1 / t2;
}
```

Listing 1: `t1` and `t2` have the same type but do not represent the same expression.

```
template<typename Fn, typename L, typename R,
         std::size_t IL, std::size_t IR>
struct Binary
{
  Binary(L const &l, R const &r);
  ...
};

template<typename L, typename R>
auto operator+(L const &l, R const &r)
-> Binary<Add, L, R, varid(l), varid(r)>
{
  return {l, r};
}
```

Listing 2: By using `varid()`, differences in expressions can be captured by incorporating the variable identity in the host node type.

3 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 2, if a new operator was defined, `varid()`, which returned the compiler's internal index of the variable given it, and both the overloaded operators and nodes types appropriately made use of `varid()`, then the actual distinction in the expressions can be captured, enabling different branches to be distinguished *and* identical branches to be matched.

Applying the change to the example in Listing 1, the left and right branches would become
`Binary<Add, double, Binary<Mul, A, B, 1, 2>, 3, 5>` and
`Binary<Add, double, Binary<Mul, A, B, 1, 2>, 4, 5>`, respectively.
Actual numbers may vary; the only point is that the generated expression types are different.

4 Design Decisions

The proposed feature can be emulated to a certain degree by wrapping variables of the same type but different value in a host class, whose type is in some way unique. Listing 3 presents a work-around to solving the problem in Listing 1, but is far from ideal as it relies on existing non-standard functionality and forcing the programmer to use an awkward syntax.

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

#define UQ(value) Unique<__COUNTER__, decltype(value)>{value}

auto evaluate(A a, B b)
{
  double c0 = 3;
  double c1 = 4;

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

  return t1 / t2;
}
```

Listing 3: `__COUNTER__` could be used as a mechanism to generate unique types.

5 Technical Specifications

`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.

```
1  double c0 = 3;
2  double c1 = 4;
3  static_assert(varid(c0) != varid(c1));
4
5  auto &cr = c0;
6  static_assert(varid(cr) == varid(c0));
7
8  cr = c1;
9  static_assert(varid(cr) == varid(c1));
```

Listing 4: Valid uses of `varid`

```
1  double c0 = 3;
2  double c1 = 4;
3  auto conexpr i0 = varid(c0 * c1); // error: expressions not supported
4  auto conexpr i1 = varid(3);       // error: literals not supported
5  auto conexpr i2 = varid(double);  // error: types not supported
```

Listing 5: Invalid uses of `varid`

6 Technical Issues

In the case of temporary variables (Listing 6) being passed to the overloaded operator functions as `const` qualified references (Listing 2), ideally the returned value of `varid` should be related to the temporary that was passed, i.e. associated to the scope from where the temporary was received rather than the local scope of the function itself. This, however, may be a difficult requirement to achieve.

```
1  auto evaluate(A a, B b)
2  {
3    auto t0 = a * b;
4    auto t1 = 3.0d + t0;
5    auto t2 = 4.0d + t0;
6
7    return t1 / t2;
8  }
```

Listing 6: `c0` and `c1` (from Listing 1) are now replaced by literals, but this should not effect the generated types of `t1` and `t2`.