Reflecting on names

Facilitating expression tree transforms

Dominic Jones

dominic.jones@gmx.co.uk

London, October 2017

Same type, different name

Write a `transform` function to yield:

```
transform(a + a) -> 2 * a


transform(a + b) -> a + b
```

where `a` and `b` are of the *same type*

Reflection

```
 123456789 123456789 123456789
           10          20          30
1
2   // main.cpp
3
4   decltype(x)*    y    =    &x;
5
```

reflect type	reflect address

Reflect location?

varid

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


// declaration of ‘l’ and ‘r’ must be visible
template<typename L, typename R>
auto operator+(L const &l, R const &r)
-> Binary<Add, L, R, varid(l), varid(r)>
{
  return {l, r};
}
```

Location stamped type

```
decltype(a + a) -> Binary<Add, T, T, 5724, 5724>


decltype(a + b) -> Binary<Add, T, T, 5724, 1396>
```

Like `sizeof`

1. keyword
2. evaluated at compile-time
3. returns an unsigned int

Like “&”, address operator

1. expects an l-value
2. returns “something like” an address:
   - hash of the file name, row and column of the referenced variable
   - traces to the referenced temporary or named variable

Valid uses

```
auto c0 = 3;
auto c1 = 4;


// compare visibly declared variables
static_assert(varid(c1) != varid(c0));


// compare with const-referenced variable
auto const &cr = c0;
static_assert(varid(cr) == varid(c0));


// compare with copied variable
auto const cc = c0;
static_assert(varid(cc) != varid(c0));
```

Invalid uses

```
auto c0 = 3;
auto c1 = 4;


// error: expressions not supported
auto constexpr i0 = varid(c0 * c1);


// error: literals not supported
auto constexpr i1 = varid(3);


// error: types not supported
auto constexpr i2 = varid(double);


// error: must be const-qualified
auto &cr = c0;
auto constexpr i3 = varid(cr);
```

Workaround

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

// using non-standard macro...
#define UQ(v) Unique<__COUNTER__, decltype(v)>{v}


auto c0 = UQ(3);
auto c1 = UQ(4);

// compare visibly declared variables
static_assert(varid(c1) != varid(c0));

...
```

Actual use

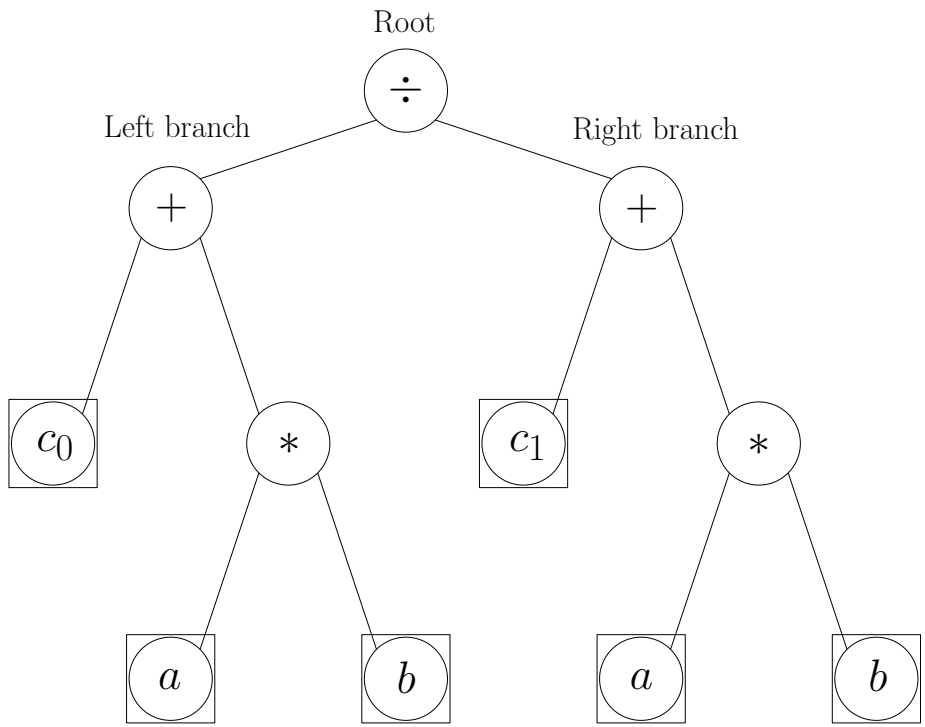
- Compile-time *Automatic Differentiation*
- Order of magnitude better performance
- Clumsy to write with `UQ` decorators

Duplicate branches?

```
auto eval(A const &a,
          B const &b)
{
  // statically identical?
  auto c0 = 3;
  auto c1 = 4;

  // 't0' evaluated twice?
  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;

  return t1 / t2;
}
```



Plan

1. Implement `varid` in C (`8cc` – github)
2. Implement it in D (`DMD` – github)
3. Write a paper about it
4. See what interest there is for C++

Temporary primitives

- Return the bit-literal value from `varid`
- Statically cast to type
- Facilitate compile-time evaluation