Infix Modifiers for Flexible Multiplication

Jim McCann*
Carnegie Mellon University

```
float a = dot(cross(
    elementwise(vec
    {1,1,0}, vec
    {1,2,3}), vec
    {0,0,1}), vec
    {1,0,0});
```

vanilla

```
float a = vec{1,1,0} *
    element() * vector
   {1,2,3} * cross() *
    vec{0,0,1} * dot()
    * vec{1,0,0};
```

infix

```
float a = vec{1,1,0}
        x vec{1,2,3}

        x vec{0,0,1}

        x vec{1,0,0};
```

by-line

Figure 1: Infix multiplication manipulators allow programmers to effortlessly switch the sense of the multiplication operator, and – with a few macros – allow a natural line-number-dependant semantics for multiplication.

Abstract

Definition of a sufficiently flexible multiplication operator remains an acknowledged grand challenge for programming languages. In this paper, we describe infix multiplication modifiers (IMMs). IMMs allow flexible, local, programmer-driven selection of multiplication operators. We demonstrate the features of our basic IMM package along with a by-line extension which allows implicit multiplication type detection.

1 Introduction

As recently observed at the prestigious and high-impact SIGBOVIK conference [McCann, 2k22], definition of a flexible and sensible multiplication operator in a modern programming language is difficult. In this paper, we describe how to push the duty of figuring out what multiplication ought to mean off on programmers instead of making a decision in a consistent way in the standard library.

This allows programmers to write simple statements like:

```
float a = vec{1,1,0} * dot() * vec
    {1,2,3};
```

Instead of the much more cumbersome:

```
float a = dot(vec{1,1,0}, vec{1,2,3});
```

Indeed, with the right macros, the infix operator itself can become implicit:

```
float a =
    vec{1,1,0}
        x vec{1,2,3};
```

How do we achieve this? Read on, dear listeners, read on.

*ix@tchow.com

2 Implementation

Our infix multiplication modifier system is designed to switch between multiplication operators effortlessly.

In our paradigm, a multiplication operator is a `struct` that provides a `::mul` static member:

```
struct dot {
    static float mul(vec const &a, vec
    const &b) {
        return a[0] * b[0]
             + a[1] * b[1]
             + a[2] * b[2];
    }
};
```

Further, a multiplication operator must be blessed by specializing a template named `has_mul`:

```
template< typename T > struct has_mul;
```

So, to indicate that `dot` is a multiplication, we write:

```
template< > struct has_mul< dot > {
    enum { value = true };
};
```

With these preliminaries out of the way we can proceed to the core of our system, which is a pair of templated overloads of `operator*`. The first matches expressions like `val * dot()` and absorbs the multiplication type into a template parameter of a wrapped value:

```
template< typename MUL, std::
    enable_if_t< has_mul< MUL >::value,
     bool > = true >
mul_vec< MUL > operator*(vec const &a,
     MUL const &m) {
```

```
    return mul_vec< MUL >{a};
}
```

Where `mul_vec< >` is a template that hangs onto a value by reference and stores a multiplication type in its type:

```
template< typename MUL >
struct mul_vec {
    vec const &wrapped;
};
```

The second `operator*` overload retrieves the reference and multiplication type from a `mul_vec` and actually performs multiplication:

```
template< typename MUL >
auto operator*(
    mul_vec< MUL > const &m,
    vec const &b
) -> decltype(MUL::mul(m.wrapped, b))
   {
    return MUL::mul(m.wrapped, b);
}
```

Notice, particularly, the use of an `auto` return type to allow deducing the return type from the particular `::mul` static member function being invoked.

2.1 Hiding Behind Macros

Of course, as many programmers know, the speed of a program is directly proportional to the number of characters in its source code. Thus, the popularity of complex type inference engines. Besides, why say what we mean when you can instead trust others to run a complex series of rules to deduce it?

Using the infix modifiers above, we can define a macro that automatically determines the type of multiplication (which, unfortunately, we need to write `x`) based on the line number:

```
#define x * indexed_mul< __LINE__ % 3
   >::type() *
```

Which, in turn, makes use of a template to fetch the proper IMM:

```
template< int I >
struct indexed_mul;
template< > struct indexed_mul< 0 > {
   using type = dot; };
template< > struct indexed_mul< 1 > {
   using type = cross; };
template< > struct indexed_mul< 2 > {
   using type = elementwise; };
```

And allows writing simple, readable code like:

```
//do not remove this comment
std::cout
    << "a * b: " << a x b << '\n'
    << "a . b: " << a x b << '\n'
    << "a x b: " << a x b << std::endl
   ;
```

Of course, making the meaning of code depend on its position in the file may seem questionable until you realize that (a) this is clearly the use case for which `#include` was designed, and (b) brittle is just another word for elegant.

3 Discussion

Though the examples in this paper are restricted to a `vec` value type, it would be straightforward to add a second template parameter to `mul_vec` (and, perhaps, change its name) to support associating any multiplication modifier with any value type.

A middleground between writing `* dot() *` and carefully checking line numbers could be found in renaming `dot` to `dot_t` and having a `constexpr dot_t dot;` available, such that `* dot *` is valid.

Note that this pattern of absorbing a computation tree into the type of an object that is eventually decayed into the computed result is not novel; it is a somewhat common trick in building (e.g.) efficient fused operations and – probably, but who cares about 'em – autodiff pipelines. As such, this joke could almost certainly go deeper, but I leave that for future work.

Acknowledgments

Early and often.

Availability

Source code for this project is available via e-mail request to the first (and only) author. It is not available publically for fear someone might actually think this is a good idea.

References

[McCann, 2k22] McCann, J. (2k22). Grand challenges in programming languages position paper: What, if anything, does multiplication even mean? In *SIGBOVIK 2k22*. http://sigbovik.org/2022/proceedings.pdf.