

MutableArithmetics

An API for mutable operations

Benoît Legat

July, 2021



Mutable sum: BigInt

Default implementation (simplified):

```
function sum(x)
  acc = zero(eltype(x))
  for el in x
    acc = acc + el
  end
  return acc
end
```

```
x = big.([2, 0, 2, 1])
```

What is `sum(x)` doing ?

Mutable sum: BigInt

Default implementation (simplified):

```
function sum(x)
  acc = zero(eltype(x))
  for el in x
    acc = acc + el
  end
  return acc
end
```

```
x = big.([2, 0, 2, 1])
```

What is `sum(x)` doing ?

```
acc = acc + el → Base.GMP.MPZ.add!(acc, el)
```

Mutable sum: symbolic

Let x be a vector of n variables/symbols (e.g. Symbolics.jl, MultivariatePolynomials.jl, JuMP.jl or MathOptInterface.jl)

```
struct Term{T}
```

```
    coef::T
```

```
    sym::Sym
```

```
end
```

```
struct Sum{T}
```

```
    terms::Vector{Term{T}}
```

```
end
```

```
Base.:+(s::Sum, t::Term) = Sum(push!(copy(s.terms), t))
```

Time complexity of `sum(x)` ?

Mutable sum: symbolic

Let x be a vector of n variables/symbols (e.g. Symbolics.jl, MultivariatePolynomials.jl, JuMP.jl or MathOptInterface.jl)

```
struct Term{T}
```

```
    coef::T
```

```
    sym::Sym
```

```
end
```

```
struct Sum{T}
```

```
    terms::Vector{Term{T}}
```

```
end
```

```
Base.:+(s::Sum, t::Term) = Sum(push!(copy(s.terms), t))
```

Time complexity of $\text{sum}(x)$? $\mathcal{O}(n^2) = \mathcal{O}(1 + 2 + \dots + n)$

Mutable sum: first solution

```
function sum(x)
    acc = zero(eltype(x))
    for el in x
        acc = MA.add!(acc, el)
    end
    return acc
end

MA.add!(a, b) = a + b # default fallback
MA.add!(a::BigInt, b::BigInt) = Base.GMP.MPZ.add!(a, b)
function MA.add!(s::Sum, t::Term)
    push!(s.terms, t)
    return s
end
```

Problem with first solution

```
struct Rational{T}
    num::T
    den::T
end

function MA.mul!(a::Rational, b::Rational)
    if # T is mutable
        MA.mul!(a.num, b.num)
        MA.mul!(a.den, b.den)
        # ... simplify with gcd ...
        return a
    else
        return a * b
    end
end
```

Explicit mutability

```
MA.mul!(a, b) = MA.operate!(*, a, b)
function MA.operate!(op, a::A, b::B) where {A, B}
    if MA.mutability(A, op, A, B) isa MA.IsMutable
        return MA.mutable_operate!(op, a, b)
    else
        return op(a, b)
    end
end
function MA.operate_to!(out::O, op, a::A, b::B) where ...
    if MA.mutability(O, op, A, B) isa MA.IsMutable
        return MA.mutable_operate_to!(out, op, a, b)
    else
        return op(a, b)
    end
end
```

Mutable API

MA.operate!/MA.operate_to! **may** mutate or not first argument. Returned value is **always** the result.

MA.mutable_operate!/MA.mutable_operate_to! mutates or **errors** (default). First argument is **always** the result.

Axiom of mutable arithmetics

```
function mutability(T::Type, op, args...)
    if mutability(T) isa IsMutable &&
        promote_operation(op, args...) == T
        return IsMutable()
    else
        return NotMutable()
    end
end
```

Implementing the API

`mutability(T)` defaults to `NotMutable()`. Hence, by default, `MA.operate!(op, args...)` redirects to `op(args...)`.

Flipping the switch

After implementing `mutability(::TypeT) = IsMutable()`, unimplemented `mutable_operate!/mutable_operate_to!` gives helpful errors.

No silent fallback to inefficient `op(args...)`!

`promote_operation` falls back to `typeof(op(zero.(args)...))`.

More **features** for the user with an API **easier** to implement.

Rewriting macro

```
MA.@rewrite(x + y * z - a * b * c) # rewritten into:  
v1 = MA.copy_if_mutable(x)  
v2 = MA.operate!(MA.add_mul, v1, y, z)  
v3 = MA.operate!(MA.sub_mul, v2, a, b, c)
```

Used in JuMP macros, critical for modeling performance.

```
MA.add_mul(x, args...) = x + *(args...)  
MA.sub_mul(x, args...) = x - *(args...)  
function MA.copy_if_mutable(x)  
    if MA.mutability(x) isa MA.IsMutable  
        return MA.mutable_copy(x)  
    end  
    return x  
end
```

Conclusion

Current implementations

BigInt, BigFloat, Rational, Polynomial,
MultivariatePolynomials, MathOptInterface, JuMP.

```
p(d) = Polynomial(big.(1:d))  
A = [p(d) for i in 1:m, j in 1:n]  
b = [p(d) for i in 1:n]  
A * b # 133 ms (2.03 M allocs: 38.012 MiB)  
MA.@rewrite(A * b) # rewritten into:  
MA.operate(*, A, b) # 8.54 ms (3.64 k allocs: 69.812 KiB)  
MA.operate!(MA.add_mul, c, A, b) # (3 allocs: 48 bytes)
```

AbstractMutable

Dispatch sum, dot and array product to MA.operate.