juliacon

MutableArithmetics: An API for mutable operations

Benoît Legat[1]

[1]Massachusetts Institute of Technology

ABSTRACT

Arithmetic operations defined in Julia do not modify their arguments. However, in many situations, a variable represents an accumulator that can be modified in-place to contain the result, e.g., when summing the elements of an array. Moreover, for types that support mutation, mutating the value may have a significant performance benefit over creating a new instance. This talk presents an interface that allows algorithms to exploit mutability in arithmetic operations in a generic manner.

Keywords

Julia, Optimization, Performance, Interface

1. Introduction

Julia enables generic algorithms that work with arbitrary number types, as long as the types implement the needed operations such as `+`, `*`, `-`, `zero`, `one`, ... The implementations of these arithmetic operations in Julia do not modify their arguments. Instead, they return a new instance of the type as the result. However, in many situations, a variable represents an accumulator that can be modified to contain the result, e.g., when summing the elements of an array or when implementing array multiplication. Moreover, for types that support mutation, mutating the value may have a significant performance benefit over creating a new instance. Examples of types that implement arithmetic operations and support mutation are `Array`s, multiple precision numbers, JuMP [3] expressions, MathOptInterface (MOI) [6] functions, and polynomials (univariate [9] or multivariate [8]).
This paper introduces an interface called MutableArithmetics. It allows mutable types to implement an arithmetic exploiting their mutability, and for algorithms to exploit their mutability while remaining completely generic. Moreover, it provides the following additional features:

(1) it re-implements part of the Julia standard library on top of the API to allow mutable types to use a more efficient version than the default one.
(2) it defines a `@rewrite` macro that rewrites an expression using the standard operations (e.g `+`, `*`, ...) into an expression that exploits the mutability of the intermediate values created when evaluating the expression.

JuMP [3] used to have its own API for mutable operations on JuMP expressions and its own JuMP-specific implementation of 1 and 2. These two features are one of the key reasons why JuMP is competitive in performance with commercial algebraic modeling languages [3, Section 3–4]. These features were refactored into MutableArithmetics, generalizing them to arbitrary mutable types. Starting from JuMP v0.21, JuMP expressions and MOI functions implement the MutableArithmetics API, and the JuMP-specific implementations of 1 and 2 were replaced by the generic versions implemented in MutableArithmetics on top of the MutableArithmetics API.

2. Design consideration

This section provides concrete examples that motivated the design of MutableArithmetics. The section is organized into four subsections that describe the need of four key features of MutableArithmetics's API.

2.1 May mutate

Consider the task of summing the elements of a vector. By default, Julia `sum` function will compute with a code equivalent to the following:

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

If the type of the elements of `x` is `BigInt`, it is more efficient to replace the line `acc = acc + el` by the line `Base.GMP.MPZ.add!(acc, el)`. Indeed, as the operation `+` cannot modify its arguments, it will need to allocate a new instance of `BigInt` to contain the result. On the other hand, `Base.GMP.MPZ.add!` modifies `acc` in place to contain the result.
Even if using `Base.GMP.MPZ.add!` provides a significant performance improvement, the time complexity order is identical: $\Theta(nm)$ in both cases where n is the number of elements and m is the number of bits of an element. We now consider a mutable element type for which exploiting mutability affects the time complexity. Consider a type |SymbolicVariable| representing a symbolic variable and the following types representing a linear combinations of these variables with coefficients of type |T|. This is examples encapsulates for instance JuMP affine expressions [3], MOI affine functions [6], polynomials (univariate [9] or multivariate [8]) or symbolic sum [5].

```
struct Term{T}
    coef::T
    sym::SymbolicVariable
end
struct Sum{T}
    terms::Vector{Term{T}}
```

```julia
end
Base.:+(s::Sum, t::Term) =
Sum(push!(copy(s.terms), t))
Base.zero(::Type{Term{T}}) where {T} =
Sum(Term{T}[])
```

Calling `sum` on a vector of n `TermT` has a time complexity $\Theta(n^2)$. Indeed, when calling `acc + el` where `acc` contains the sum of the first `k` terms and `el` is the $(k+1)$th term, the result cannot mutate `acc.terms` and the copy of `acc.terms` has time complexity $\Theta(k)$. A possible mutable interface would be to define an `add!` function that is similar to + with the only difference being that it is allowed to modify its first argument. By default, `add!` would fall back to calling + so that a method calling `add!` would both exploit the mutability of mutable types but would also work for non-mutable types. For our example, an implementation could be:

```julia
function sum(x)
    acc = zero(eltype(x))
    for el in x
        acc = add!!(acc, el)
    end
    return acc
end
add!!(a, b) = a + b # default fallback
add!!(a::BigInt, b::BigInt) =
Base.GMP.MPZ.add!(a, b)
function add!!(s::Sum, t::Term)
    push!(s.terms, t)
    return s
end
```

Note that the time complexity of the sum of n `Term` is now $\Theta(n)$. Julia implements a specialized method for computing the sum of `BigInt`s that uses `Base.GMP.MPZ.add!`. Similarly, before its version v0.21, JuMP used to implement a specialized method for the sum of JuMP expressions. The advantage of having a standardized API for mutable addition is that only one implementation of `sum` is needed. This approach of an API based on a function that may mutate its first argument in order to allow the same code to work both for mutable and non-mutable type is used by the `!!` convention in BangBang [1], the mutable API in AbstractAlgebra [4], as well as the `destructive_add!` function in JuMP v0.20.

2.2 Should mutate

When writing a code that should mutate the first argument of an operation, an API that silently returns the result without modifying the first argument is not appropriate.
To motivate this, consider the rational Julia type:

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

Suppose we want to mutate[1] some rational `a::Rational` to the product of `a::Rational` and some other rational `b::Rational` (ignoring the simplification with `gcd` for simplicity).
Using `a.num = mul!!(a.num, b.num)` and `a.den = mul!!(a.den, b.den)` (where `mul!!` follows BangBang's convention) is not an option since the `Rational` struct is not mutable.

For this reason, there are also mutable operations that should mutate the first argument. This is the approach used by the `!` convention in Julia as well as the `add_to_expression!` in JuMP.

2.3 Mutability

A third useful feature for users of a mutable API is the ability to determine whether objects of a given type can be mutated[1] to the result of a mutable operation. To motivate this, consider again the multiplication of rational number introduces in the previous section. An implementation `mul!!` (where `mul!!` may mutate its first argument and `mul!` should mutate its first argument) for rational numbers could be:

```julia
function mul!!(a::Rational{S}, b::Rational{T})
    if # S can be mutated to `*(::S, ::T)`
        mul!(a.num, b.num)
        mul!(a.den, b.den)
        return a
    else
        return a * b
    end
end
```

This third feature would be needed to implement this `if` clause.

2.4 Promotion

Algorithms that can exploit mutability often start by creating an accumulator of an appropriate type.
Consider the following matrix-vector multiplication implementation with `SymbolicVariable` where `mul_to!` mutates[1] `c` to `A * b`.

```julia
function Base.:*(A::Matrix{S}, b::Vector{T})
    c = Vector{U}(undef, size(A, 1)) # What is T ?
    return mul_to!(c, A, b)
end
```

What should be the element type `U` of the accumulator `c` ? For instance, if `S` is `Float64` and `T` is `SymbolicVariable` then `U` should be `Sum{Float64}`. LinearAlgebra uses `Base.promote_op` for this which relies on Julia inference to determine the type of a sum of products of elements of type `S` and `T`.
In the summing example introduced in section 2.1, the type of the accumulator should also be determined as the type of the sum of elements of the vector. For the `sum` function, Julia uses `zero` as it is defined as the additive identity element.

3. Implementing the interface

MutableArithmetics defines the following four functions that provides the features motivated in the corresponding four subsections of the previous section.

(1) `operate!!(op::Function, args...)` (resp. `operate_to!!(output, op::Function, args...)`) returns the result of `op(args...)` and may mutate `args[1]` (resp. `output`).
(2) `operate!(op::Function, args...)` (resp. `operate_to!(output, op::Function, args...)`) mu-

[1] In this paper, the terminology "mutate x to y" means mutating x in such a way that its value after the mutation is equal to y

tate[1] `args[1]` (resp. `output`) to the result of `op(args...)` and returns it.

(3) `mutability(T::Type, op::Function, args::Type...)` is a trait returning `IsMutable()` if objects of type `T` can be mutated[1] to the result of `op(::args[1], ::args[2], ...)` and `IsNotMutable()` otherwise.

(4) `promote_operation(op::Function, args::Type...)` returns the return type of `op(::args[1], ::args[2], ...)`.

As we detailed in the previous section, this API covers many use cases. The downside of such a varied API is that it seems to be a lot of work to implement it for a mutable type. We show in the remainder of this section how the MutableArithmetics API remains simple to implement nevertheless.

3.1 Promotion fallback

First, `promote_operation` can have default fallback. For instance, `promote_operation(+, ::Type{S}, ::Type{T})` defaults to `typeof(zero(S) + zero(T))` which is correct if `+(::S, ::T)` is type-stable. As the result of `promote_operation` only depends on the signature of the function,
There are two cases for which this default implementation of `promote_operation` is not sufficient. As we will see below, `promote_operation` is a the core of many operation so it is important that it is efficient. Julia may be able to compute the result of `typeof(zero(S) + zero(T))` at compile time. However, if the body of `promote_operation` is not evaluated at compile-time, this can cause performance issue. This is amplified for mutable types as `zero(S) + zero(T)` may allocate. Moreover, if `zero(S) + zero(T)` ends up calling `promote_operation(+, S, T)`, this default implementation will not terminate. In both of these cases, `promote_operation` should have a specialized implementation, e.g., by hardcoding the result for each pairs of concrete types `S` and `T`. Note that implementing `promote_operation` should be easier than implementing the actual operation where the actual value of the result need to be computed, not just the type so this should not consitute a burden for the implementation.

3.2 May mutate fallback

We have the following default implementations of `operate!!` (resp. `operate_to!!`).

```
function operate!!(op, args...)
    T = typeof.(args)
    if mutability(T[1], op, T...) isa IsMutable
        return operate!(op, args...)
    else
        return op(args...)
    end
end
function operate_to!!(output, op, args...)
    O = typeof(output)
    T = typeof.(args)
    if mutability(O, op, T...) isa IsMutable
        return operate_to!(output, op, args...)
    else
        return op(args...)
    end
end
```

Note that this default implementation should have optimal performance in case the trait `mutability` is optimized out by the compiler. Indeed, as the functions `op` and `operate!(op, ...)` (resp. `operate_to!(op, ...)`) are strictly more specific than `operate!!` (resp. `operate_to!!`), if the run-time cost of the trait `mutability` is zero, then if a specialized method is faster than this implementation, it means that either `op` or `operate!(op, ...)` (resp. `operate_to!(op, ...)`) can be implemented more efficiently.

3.3 Mutability fallback

It turns out that all types considered at the moment fall into two categories. The first category is made of the types `T` for which `mutability(T, ...)` always return `IsNotMutable()`. These are typically the non-mutable type, e.g., `Int`, `Float64`, `Rational{Int}`, ... In the second category are the types `T` for which `mutability(T, op, args...)` returns `IsMutable()` if and only if `T == promote_operation(op, args...)`. Based on this observation, we define `mutability(T::Type)` which returns `IsMutable()` if `T` is in the first category and `IsNotMutable()` if `T` is in the second category. Then we have the following fallback for `mutability`:

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

3.4 Minimal interface

In summary, for a type `Foo` to implement the interface, the following line should be implemented:

```
mutability(::Type{Foo}) = IsMutable()
```

as well as the following lines for each operation (let's assume the operation is + and the result type is `Foo`),

```
promote_operation(::typeof(+), ::Type{Foo}, ::Type{Foo}) =
Foo
function operate!(::typeof(+), a::Foo, b::Foo)
    # ...
    return a
end
function operate_to!(output::Foo, ::typeof(+), a::Foo, b::F
    # ...
    return output
end
```

Then

```
mutability(::Foo, +, Foo, Foo),
operate!!(+, ::Foo, ::Foo),
operate_to!!(::Foo, +, ::Foo, ::Foo),
add!(::Foo, ::Foo),
add_to!(::Foo, ::Foo, ::Foo),
```

`add!!(::Foo, ::Foo)` and
`add_to!!(::Foo, ::Foo, ::Foo)`
will be available as well for the user thanks to the default fallbacks.

4. Rewriting macro

As mentioned in the introduction, MutableArithmetics implements a `@rewrite` macro that rewrites:

```
@rewrite(a * b + c * d - e * f * g - sum(i *
y[i]^2 for i in 2:n))
```

into

```
acc0 = Zero()
acc1 = add_mul!!(acc0, a, b)
acc2 = add_mul!!(acc1, c, d)
acc3 = sub_mul!!(acc2, e, f, g)
for i in 2:n
  acc3 = sub_mul!!(acc3, i, y[i])
end
acc3
```

where

```
add_mul(x, args...) = x + *(args...)
sub_mul(x, args...) = x - *(args...)
```

The code produced by the `@rewrite` macro does not assume that any of the objects `a`, `b`, ... can be mutated. However, it exploits the mutability of the intermediate expressions `acc0`, `acc1`, `acc2` and `acc3`. Note that different accumulator variables are used because the type of the accumulator may change.

5. Benchmarks and buffers

In this section, we provide a benchmark and illustrate how MutableArithmetics allows to preallocate buffers needed by low-level operations.

5.1 Matrix-vector product

Consider the product between a matrix and a vector of `BigInt`s. `LinearAlgebra.mul!` uses a generic implementation that does not exploit the mutability of `BigInt`s. We can see in the following benchmark [2] that more that 3 MB are allocated.

```
n = 200
l = big(10)
A = rand(-l:l, n, n)
b = rand(-l:l, n)
c = zeros(BigInt, n)

using BenchmarkTools
import LinearAlgebra
@benchmark LinearAlgebra.mul!($c, $A, $b)

# output

 Time  (median):      5.900 ms
 Time  (mean):       12.286 ms
 Memory: 3.66 MiB, allocs: 197732.
```

The generic implementation in MutableArithmetics exploits the mutability of the elements of `c`. This provides a significant speedup and a drastic reduction of memory usage:

```
@benchmark add_mul!($c, $A, $b)

# output

 Time  (median):       1.001 ms
 Time  (mean):         1.021 ms
 Memory: 48 bytes, allocs: 3.
```

In fact, it also exploits the mutability of the intermediate terms. If the generic implementation was calling

```
operate!(add_mul, c[i], A[i, j], b[j])
```

it would allocate a `BigInt` to hold an intermediate value as in:

```
tmp = A[i, j] * b[j]
operate!(+, c[i], tmp)
```

In order to avoid allocating n^2 new `BigInt`s, MutableArithmetics enables operations to communicate the buffers they need to allocate through the `buffer_for` function. The buffer can then be reused between multiple occurence of the same operation with `buffered_operate!`. By default, `buffer_for` returns `nothing` and `buffered_operate!` has the following fallback:

```
buffered_operate!(::Nothing, args...) =
operate!(args...)
```

`BigInt` can allow the buffer to be reused by implementing:

```
buffer_for(::add_mul, ::Type{BigInt}...) =
BigInt()
function operate!(::add_mul, a::BigInt, b::BigInt,
                  c::BigInt)
    buffered_operate!(BigInt(), ::add_mul, a, b, c)
end
```

Then, the matrix multiplication can create the buffer only once and then call

```
buffered_operate!(buf, add_mul, c[i], A[i, j], b[j])
```

This explains why there is only 48 bytes allocated, this is the allocation of a single `BigInt()`.
In fact, a buffer needed for a low-level operation can even be communicated at the level of higher level operations. This allows for instance to allocate the buffer only once even if several matrix products are computed:

```
buf = buffer_for(
    add_mul, typeof(c), typeof(A), typeof(b))
@allocated buffered_operate!(buf, add_mul, c, A, b)

# output
```

```
0
```

5.2 Mutability layers

Mutable objects may have multiple mutable layers. It is paramount for mutability API to allow the user to exploit the mutability from the top layer to the bottom layer. Consider the following example using Polynomials [9].

```
using Polynomials
p(d) = Polynomial(big.(1:d))
z(d) = Polynomial([zero(BigInt) for i in 1:d])
A = [p(d) for i in 1:m, j in 1:n]
b = [p(d) for i in 1:n]
c = [z(2d - 1) for i in 1:m]
```

The arrays contain 3 layers of mutability: `Array`, `Polynomial` and `BigInt`. As shown in the benchmark below, impact on performance is amplified by the number of layers.

```
julia> @benchmark LinearAlgebra.mul!($c, $A, $b)
 Time  (median):      131.901 ms
 Time  (mean):        128.542 ms
 Memory: 38.02 MiB, allocs: 2032580.

julia> @benchmark add_mul!($c, $A, $b)
 Time  (median):      7.633 ms
 Time  (mean):        7.687 ms
 Memory: 48 bytes, allocs: 3.

julia> buf = buffer_for(
    add_mul, typeof(c), typeof(A), typeof(b))
0

julia> @allocated buffered_operate!(
           buf, add_mul, c, A, b)
0
```

As a matter of fact, one of the motivation for MutableArithmetics was to improve the performance of SumOfSquares [7]. SumOfSquares was using multivariate polynomials with JuMP expressions or MOI functions as coefficients. JuMP had a interface for exploiting the mutability of its expressions but MultivariatePolynomials was not exploiting it. MultivariatePolynomials now implements MutableArithmetics and also exploits the mutability of its coefficients, whether they are `BigInt`, JuMP expressions, MOI functions or any other type implementing MutableArithmetics.

6. Conclusion

MutableArithmetics provides an interface for mutable operations. As detailed in this paper, the design of the interface provides both an extensive set of features for the user without sacrificing the ease of implementation of the interface. Moreover, it provides a zero-cost abstraction so that a single generic implementation can handle mutable and non-mutable inputs. As the same API is used for arrays, functions, numbers, ... multi-layered mutability can be exploited efficiently and the intermediate allocations needed by inner layers can be preallocated from the outside layers using a buffer API.

7. References

[1] Takafumi Arakaki. JuliaFolds/BangBang.jl: v0.3.31.

[2] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv e-prints*, Aug 2016. 1608.04295.

[3] Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[4] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 157–164, New York, NY, USA, 2017. ACM. doi:10.1145/3087604.3087611.

[5] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwozdz, Viral B Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *arXiv preprint arXiv:2105.03949*, 2021.

[6] Benoît Legat, Oscar Dowson, Joaquim Dias Garcia, and Miles Lubin. MathOptInterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, 2021. In press.

[7] Benoît Legat and Tillmann Weisser. Sumofsquares: A julia package for polynomial optimization. In *INFORMS Annual Meeting*, 2020.

[8] Benoît Legat, Sascha Timme, and Robin Deits. JuliaAlgebra/MultivariatePolynomials.jl: v0.3.18, July 2021. doi:10.5281/zenodo.5083796.

[9] John Verzani, Miles Lucas, Zdeněk Hurák, and Jameson Nash. JuliaMath/Polynomials.jl: v2.0.14.