What's wrong with this julia?

Kenta Sato (佐藤建太)

Twitter/GitHub: @bicycle1885

A Sad Story

Is Julia as fast as C/C++? Where is the download link!?

Oh, the syntax looks really neat!

Ok, it works. Now let's compare it with my Python script.

Huh? not so fast as he said, maybe comparable with Python?

Julia? Yes, I tried it last month but blahblah...

A Sad Story

Is Julia as fast as C/C++? Where is the download link!?

Oh, the syntax looks really neat!

Ok, it works. Now let's compare it with my Python script.

Huh? not so fast as he said, maybe comparable with Python?

Julia? Yes, I tried it last month but blahblah...

Prob(their code does something wrong | newbies complain about performance) > 0.95

We are going to learn very basic skills to write (nearly) optimal Julia code from three cases.

Case 1

Julia

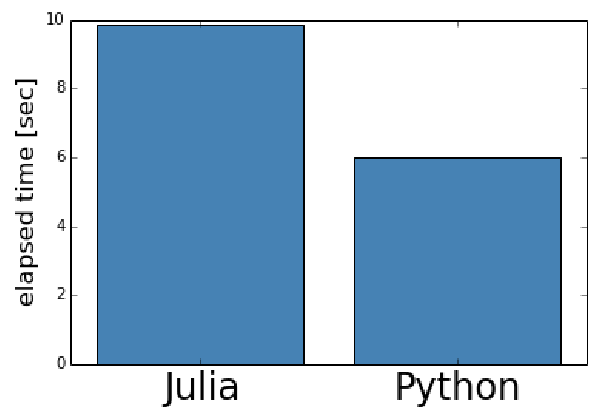
```
s = 0
n = 50000000
for i in 1:n
    s += i
end
println(s)
```

```
$ time julia case1.jl
real    0m9.859s
...
```

Python

```
s = 0
n = 50000000
for i in xrange(1, n + 1):
    s += i
print(s)
```

```
$ time python case1.py
real    0m5.998s
...
```



Is Julia slower than Python? Really??

Case 1

Do Not Write a Heavy Loop at the Top Level!

- The top-level code is interpreted, not compiled.
- Top level is not intended to run numeric computation.
- Should be written in a `function` / `let` block.

Julia

```julia
s = 0
n = 50000000
for i in 1:n
    s += i
end
println(s)
```

```
$ time julia case1.jl
real    0m9.859s
...
```

Julia

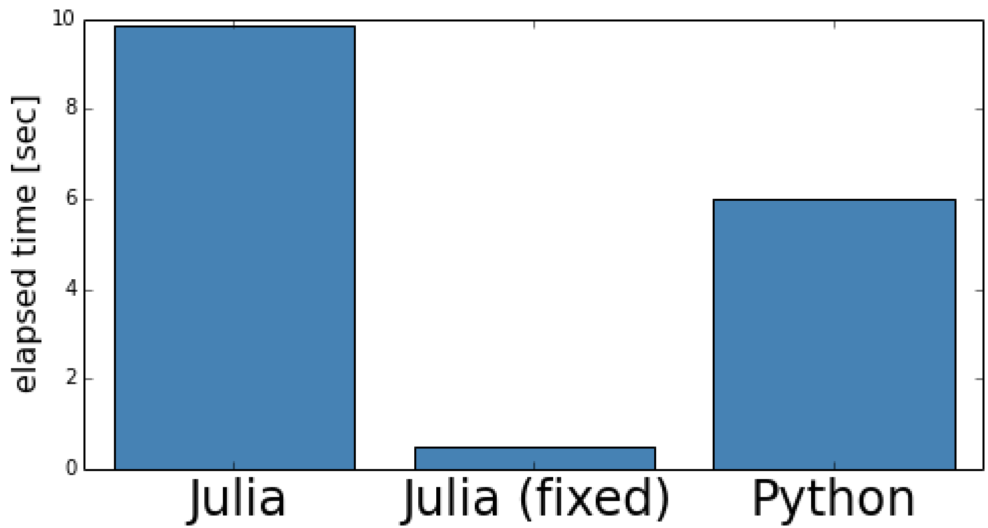
```julia
let
    s = 0
    n = 50000000
    for i in 1:n
        s += i
    end
    println(s)
end
```

```
$ time julia case1.fix.jl
real    0m0.494s
...
```

Case 1

Do Not Write a Heavy Loop at the Top Level!

- The top-level code is interpreted, not compiled.
- Top level is not intended to run numeric computation.
- Should be written in a `function`/`let` block.



9.859s / 0.494s = 20x faster

Case 2

Julia

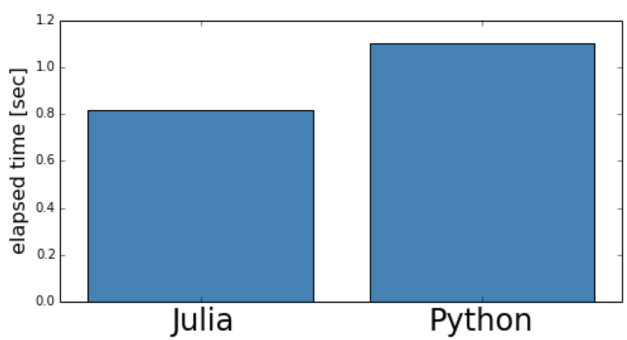
```
function dot(x, y)
    s = 0
    for i in 1:length(x)
        s += x[i] * y[i]
    end
    s
end

n = 10000000
x = zeros(n)
y = zeros(n)
dot(x[1:3], y[1:3])  # compile
```

```
$ julia -L case2.jl -e '@time do
elapsed time: 0.814667538 second
```

Python

```
def dot(x, y):
    s = 0
    for i in xrange(len(x)):
        s += x[i] * y[i]
    return s

n = 10000000
x = [0] * n
y = [0] * n
```

```
% timeit dot(x, y)
1 loops, best of 3: 1.1 s per lo
```



Case 2

Make Variable Types Stable!

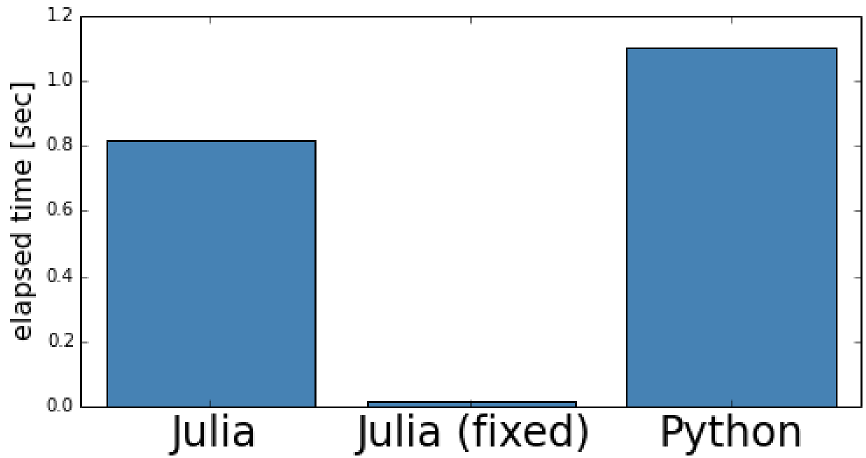
- Literal `0` is typed as `Int`.
- Hence Julia assumes the type of the variable `s` is `Int`.
- The element type of `x` and `y` is `Float64`.
- In Julia, `Int + Float64 * Float64` is `Float64`.
- `s += x[i] * y[i]` disturbs the type stability of `s`.

```julia
function dot(x, y)
    s = 0
    for i in 1:length(x)
        s += x[i] * y[i]
    end
    s
end

...
```

```julia
function dot(x, y)
    s = zero(eltype(x))
    for i in 1:length(x)
        s += x[i] * y[i]
    end
    s
end

...
```

Case 2

- Type stability of variables dramatically improves the performance.



- Much less memory allocation and GC time

```
$ julia -L case2.jl -e '@time dot(x, y)'
elapsed time: 0.814667538 seconds (320013880 bytes allocated, 17.5
```

```
$ julia -L case2.fix.jl -e '@time dot(x, y)'
elapsed time: 0.017035491 seconds (13896 bytes allocated)
```

0.814667538s / 0.017035491s = 48x faster

Case 2

- The emitted LLVM code is cleaner when types are stable.

```
In [142]: !julia -L case2.jl -e '@code_llvm dot(x, y)'

define %jl_value_t* @julia_dot_19813(%jl_value_t*, %jl_value_t**, i32) {
top:
  %3 = alloca [5 x %jl_value_t*], align 8
  %.sub = getelementptr inbounds [5 x %jl_value_t*]* %3, i64 0, i64 0
  %4 = getelementptr [5 x %jl_value_t*]* %3, i64 0, i64 2, !dbg !287
  store %jl_value_t* inttoptr (i64 6 to %jl_value_t*), %jl_value_t** %.sub, align 8
  %5 = load %jl_value_t*** @jl_pgcstack, align 8, !dbg !287
  %6 = getelementptr [5 x %jl_value_t*]* %3, i64 0, i64 1, !dbg !287
  %.c = bitcast %jl_value_t** %5 to %jl_value_t*, !dbg !287
  store %jl_value_t* %.c, %jl_value_t** %6, align 8, !dbg !287
  store %jl_value_t** %.sub, %jl_value_t*** @jl_pgcstack, align 8, !dbg !287
  store %jl_value_t* null, %jl_value_t** %4, align 8, !dbg !287
  %7 = getelementptr [5 x %jl_value_t*]* %3, i64 0, i64 3
  store %jl_value_t* null, %jl_value_t** %7, align 8
  %8 = getelementptr [5 x %jl_value_t*]* %3, i64 0, i64 4
  store %jl_value_t* null, %jl_value_t** %8, align 8
  %9 = load %jl_value_t** %1, align 8, !dbg !287
  %10 = getelementptr %jl_value_t** %1, i64 1, !dbg !287
  %11 = load %jl_value_t** %10, align 8, !dbg !287
  store %jl_value_t* inttoptr (i64 140727623488064 to %jl_value_t*), %jl_value_t** %4, align
8, !dbg !288
  %12 = getelementptr inbounds %jl_value_t* %9, i64 2, !dbg !289
  %13 = bitcast %jl_value_t* %12 to i64*, !dbg !289
  %14 = load i64* %13, align 8, !dbg !289, !tbaa %jtbaa_arraylen
  %15 = icmp sgt i64 %14, 0, !dbg !289
  br i1 %15, label %L.preheader, label %L5, !dbg !289

L.preheader:                                      ; preds = %top
  %16 = getelementptr inbounds %jl_value_t* %11, i64 2, !dbg !294
  %17 = bitcast %jl_value_t* %16 to i64*, !dbg !294
  %18 = getelementptr inbounds %jl_value_t* %9, i64 1, !dbg !294
  %19 = bitcast %jl_value_t* %18 to i8**, !dbg !294
  %20 = getelementptr inbounds %jl_value_t* %11, i64 1, !dbg !294
  %21 = bitcast %jl_value_t* %20 to i8**, !dbg !294
  br label %L, !dbg !289

L:                                                ; preds = %L.preheader, %idxend2
  %22 = phi %jl_value_t* [ %44, %idxend2 ], [ inttoptr (i64 140727623488064 to %jl_value_t*),
%L.preheader ]
  %"#s2.0" = phi i64 [ %30, %idxend2 ], [ 1, %L.preheader ]
  store %jl_value_t* %22, %jl_value_t** %7, align 8, !dbg !294
  %23 = add i64 %"#s2.0", -1, !dbg !294
  %24 = load i64* %13, align 8, !dbg !294, !tbaa %jtbaa_arraylen
  %25 = icmp ult i64 %23, %24, !dbg !294
  br i1 %25, label %idxend, label %oob, !dbg !294

oob:                                              ; preds = %L
  %26 = load %jl_value_t** @jl_bounds_exception, align 8, !dbg !294, !tbaa %jtbaa_const
  call void @jl_throw_with_superfluous_argument(%jl_value_t* %26, i32 4), !dbg !294
  unreachable, !dbg !294

idxend:                                           ; preds = %L
  %27 = load i64* %17, align 8, !dbg !294, !tbaa %jtbaa_arraylen
  %28 = icmp ult i64 %23, %27, !dbg !294
  br i1 %28, label %idxend2, label %oob1, !dbg !294

oob1:                                             ; preds = %idxend
  %29 = load %jl_value_t** @jl_bounds_exception, align 8, !dbg !294, !tbaa %jtbaa_const
  call void @jl_throw_with_superfluous_argument(%jl_value_t* %29, i32 4), !dbg !294
  unreachable, !dbg !294

idxend2:                                          ; preds = %idxend
  %30 = add i64 %"#s2.0", 1, !dbg !289
  %31 = load i8** %19, align 8, !dbg !294, !tbaa %jtbaa_arrayptr
  %32 = bitcast i8* %31 to double*, !dbg !294
  %33 = getelementptr double* %32, i64 %23, !dbg !294
  %34 = load double* %33, align 8, !dbg !294, !tbaa %jtbaa_user
  %35 = load i8** %21, align 8, !dbg !294, !tbaa %jtbaa_arrayptr
  %36 = bitcast i8* %35 to double*, !dbg !294
  %37 = getelementptr double* %36, i64 %23, !dbg !294
  %38 = load double* %37, align 8, !dbg !294, !tbaa %jtbaa_user
  %39 = fmul double %34, %38, !dbg !294
  %40 = call %jl_value_t* @alloc_2w(), !dbg !294
  %41 = getelementptr inbounds %jl_value_t* %40, i64 0, i32 0, !dbg !294
  store %jl_value_t* inttoptr (i64 140727623414720 to %jl_value_t*), %jl_value_t** %41, align
8, !dbg !294
  %42 = getelementptr inbounds %jl_value_t* %40, i64 1, i32 0, !dbg !294
  %43 = bitcast %jl_value_t** %42 to double*, !dbg !294
  store double %39, double* %43, align 8, !dbg !294
  store %jl_value_t* %40, %jl_value_t** %8, align 8, !dbg !294
  %44 = call %jl_value_t* @jl_apply_generic(%jl_value_t* inttoptr (i64 140727658110176 to %j
l_value_t*), %jl_value_t** %7, i32 2), !dbg !294
  store %jl_value_t* %44, %jl_value_t** %4, align 8, !dbg !294
  %45 = icmp eq i64 %"#s2.0", %14, !dbg !294
  br i1 %45, label %L5, label %L, !dbg !294

L5:                                               ; preds = %idxend2, %top
  %46 = phi %jl_value_t* [ inttoptr (i64 140727623488064 to %jl_value_t*), %top ], [ %44, %id
xend2 ]
  %47 = load %jl_value_t** %6, align 8, !dbg !298
  %48 = getelementptr inbounds %jl_value_t* %47, i64 0, i32 0, !dbg !298
  store %jl_value_t** %48, %jl_value_t*** @jl_pgcstack, align 8, !dbg !298
  ret %jl_value_t* %46, !dbg !298
}
```

Type-unstable code

```
In [143]: !julia -L case2.fix.jl -e '@code_llvm dot(x, y)'

define double @julia_dot_19813(%jl_value_t*, %jl_value_t*) {
top:
  %2 = getelementptr inbounds %jl_value_t* %0, i64 2, !dbg !287
  %3 = bitcast %jl_value_t* %2 to i64*, !dbg !287
  %4 = load i64* %3, align 8, !dbg !287, !tbaa %jtbaa_arraylen
  %5 = icmp sgt i64 %4, 0, !dbg !287
  br i1 %5, label %L.preheader, label %L5, !dbg !287

L.preheader:                                      ; preds = %top
  %6 = load i64* %3, align 8, !dbg !292, !tbaa %jtbaa_arraylen
  %7 = getelementptr inbounds %jl_value_t* %1, i64 2, !dbg !292
  %8 = bitcast %jl_value_t* %7 to i64*, !dbg !292
  %9 = getelementptr inbounds %jl_value_t* %0, i64 1, !dbg !292
  %10 = bitcast %jl_value_t* %9 to i8**, !dbg !292
  %11 = getelementptr inbounds %jl_value_t* %1, i64 1, !dbg !292
  %12 = bitcast %jl_value_t* %11 to i8**, !dbg !292
  br label %L, !dbg !287

L:                                                ; preds = %L.preheader, %idxend2
  %"#s2.0" = phi i64 [ %19, %idxend2 ], [ 1, %L.preheader ]
  %s.0 = phi double [ %29, %idxend2 ], [ 0.000000e+00, %L.preheader ]
  %13 = add i64 %"#s2.0", -1, !dbg !292
  %14 = icmp ult i64 %13, %6, !dbg !292
  br i1 %14, label %idxend, label %oob, !dbg !292

oob:                                              ; preds = %L
  %15 = load %jl_value_t** @jl_bounds_exception, align 8, !dbg !292, !tbaa %jtbaa_const
  call void @jl_throw_with_superfluous_argument(%jl_value_t* %15, i32 4), !dbg !292
  unreachable, !dbg !292

idxend:                                           ; preds = %L
  %16 = load i64* %8, align 8, !dbg !292, !tbaa %jtbaa_arraylen
  %17 = icmp ult i64 %13, %16, !dbg !292
  br i1 %17, label %idxend2, label %oob1, !dbg !292

oob1:                                             ; preds = %idxend
  %18 = load %jl_value_t** @jl_bounds_exception, align 8, !dbg !292, !tbaa %jtbaa_const
  call void @jl_throw_with_superfluous_argument(%jl_value_t* %18, i32 4), !dbg !292
  unreachable, !dbg !292

idxend2:                                          ; preds = %idxend
  %19 = add i64 %"#s2.0", 1, !dbg !287
  %20 = load i8** %10, align 8, !dbg !292, !tbaa %jtbaa_arrayptr
  %21 = bitcast i8* %20 to double*, !dbg !292
  %22 = getelementptr double* %21, i64 %13, !dbg !292
  %23 = load double* %22, align 8, !dbg !292, !tbaa %jtbaa_user
  %24 = load i8** %12, align 8, !dbg !292, !tbaa %jtbaa_arrayptr
  %25 = bitcast i8* %24 to double*, !dbg !292
  %26 = getelementptr double* %25, i64 %13, !dbg !292
  %27 = load double* %26, align 8, !dbg !292, !tbaa %jtbaa_user
  %28 = fmul double %23, %27, !dbg !292
  %29 = fadd double %s.0, %28, !dbg !292
  %30 = icmp eq i64 %"#s2.0", %4, !dbg !292
  br i1 %30, label %L5, label %L, !dbg !292

L5:                                               ; preds = %idxend2, %top
  %s.1 = phi double [ 0.000000e+00, %top ], [ %29, %idxend2 ]
  ret double %s.1, !dbg !296
}
```

Type-stable code

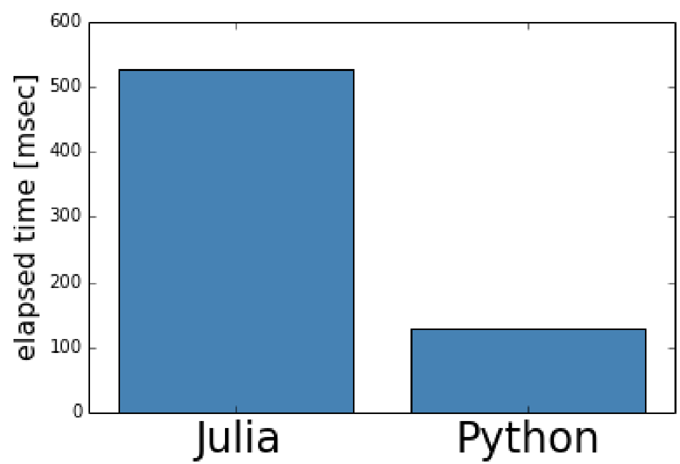
Case 3

Julia

```julia
...
function optimize(x0, n_iter)
    η = 0.1
    x = copy(x0)
    g = zeros(length(x0))
    for i in 1:n_iter
        grad!(g, x)
        x -= η * g
    end
    x
end
```

Python

```python
...
def optimize(x0, n_iter):
    eta = 0.1
    x = np.copy(x0)
    g = np.zeros(len(x0))
    for i in xrange(n_iter):
        set_grad(g, x)
        x -= eta * g
    return x
```



Case 3

`-=` for array types is not an in-place operator!

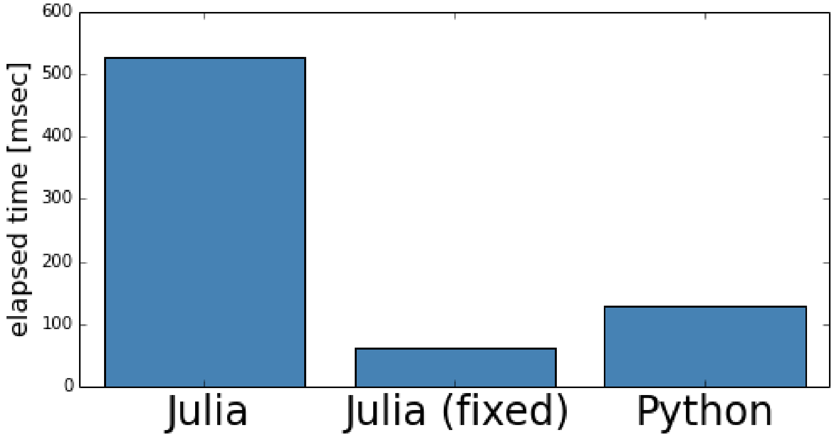
- `x -= η * g` works like:
  - `y = η * g`: product
  - `z = x - y`: subtract
  - `x = z`: bind
- For each iteration:
  - Create two arrays (`y` and `z`) and
  - Discard two arrays (`x` and `y`)
- Lots of memories are wasted!
  - 53% of time is consumed in GC!

```
$ julia -L case3.jl -e '@time optimize(x0, 50000)'
elapsed time: 0.527045846 seconds
    (812429992 bytes allocated, 53.60% gc time)
```

Case 3

```julia
...
function optimize(x0, n_iter)
    ...
    for i in 1:n_iter
        grad!(g, x)
        x -= η * g
    end
    x
end
```

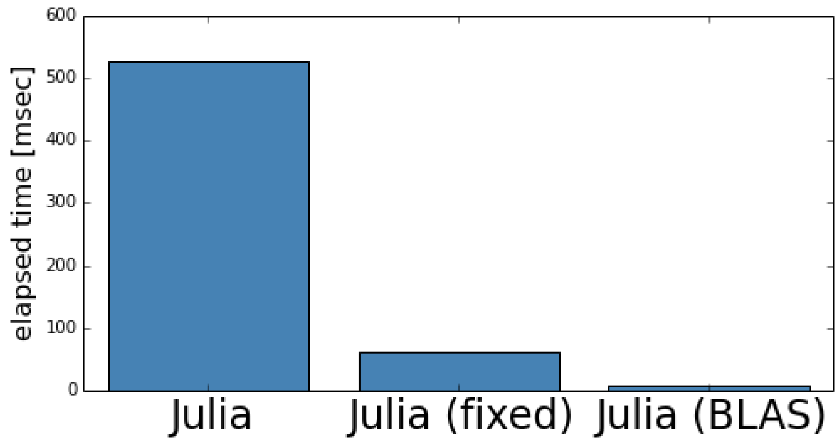
```julia
...
function optimize(x0, n_iter)
    ...
    for i in 1:n_iter
        grad!(g, x)
        for j in 1:n
            x[j] -= η * g[j]
        end
    end
    x
end
```



Case 3

```julia
...
function optimize(x0, n_iter)
    ...
    for i in 1:n_iter
        grad!(g, x)
        for j in 1:n
            x[j] -= η * g[j]
        end
    end
    x
end
```

```julia
...
function optimize(x0, n_iter)
    ...
    for i in 1:n_iter
        grad!(g, x)
        BLAS.scal!(n, η, g, 1)
    end
    x
end
```



Take-home Points

- Julia is definitely fast unless you do it wrong.
- You should know what code is *smooth* in Julia.
- Manual/Performance Tips is a definitive guide to know the smoothness of Julia.
- Utilize standard tools to analyze bottlenecks:
  - `@time` macro
  - `@profile` macro
  - `--track-allocation` option
  - `@code_typed` / `@code_lowered` / `@code_llvm` / `@code_native` macros (from higher level to lower)
  - `@code_warntype` (v0.4)