7 Julia Gotchas and How to Handle Them

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the
WINNOWER

COMPUTER SCIENCES

7 Julia Gotchas and How to Handle Them

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
August 18, 2018

DOI:
10.15200/winn.153459.99353

ARCHIVED:
August 18, 2018

CITATION:
Christopher Rackauckas, 7 Julia Gotchas and How to Handle Them, *The Winnower* 5:e153459.99353 , 2018 , DOI: 10.15200/winn.153459.99353

Let me start by saying Julia is a great language. I love the language, it is what I find to be the most powerful and intuitive language that I have ever used. It's undoubtedly my favorite language. That said, there are some "gotchas", tricky little things you need to know about. Every language has them, and one of the first things you have to do in order to master a language is to find out what they are and how to avoid them. The point of this blog post is to help accelerate this process for you by exposing some of the most common "gotchas" offering alternative programming practices.

Julia is a good language for understanding what's going on because there's no magic. The Julia developers like to have clearly defined rules for how things act. This means that all behavior can be explained. However, this might mean that you need to think about what's going on to understand why something is happening. That's why I'm not just going to lay out some common issues, but I am also going to explain why they occur. You will see that there are some very similar patterns, and once you catch onto the patterns, you will not fall for any of these anymore. Because of this, there's a slightly higher learning curve for Julia over the simpler languages like MATLAB/R/Python. However, once you get the hang of this, you will fully be able to use the conciseness of Julia while obtaining the performance of C/Fortran. Let's dig in.

GOTCHA #1: THE REPL (TERMINAL) IS THE GLOBAL SCOPE

For anyone who is familiar with the Julia community, you know that I have to start here. This is by far the most common problem reported by new users of Julia. Someone will go "I heard Julia is fast!", open up the REPL, quickly code up some algorithm they know well, and execute that script. After it's executed they look at the time and go "wait a second, why is this as slow as Python?"

Because this is such an important issue and pervasive, let's take some extra time delving into why this happens so we understand how to avoid it.

SMALL INTERLUDE INTO WHY JULIA IS FAST

To understand what just happened, you have to understand that Julia is about not just code compilation, but also type-specialization (i.e. compiling code which is specific to the given types). Let me repeat: Julia is not fast because the code is compiled using a JIT compiler, rather it is fast because type-specific code is compiled and ran.

If you want the full story, checkout some of the notes I've written for an upcoming workshop. I am going to summarize the necessary parts which are required to understand why this is such a big deal.

Type-specificity is given by Julia's core design principle: multiple dispatch. When you write the code:

```
function f(a,b)
  return 2a+b
end
```

you may have written only one "function", but you have written a very large amount of "methods". In Julia parlance, a function is an abstraction and what is actually called is a method. If you call f(2.0,3.0), then Julia will run a compiled code which takes in two floating point numbers and returns the value 2a+b. If you call f(2,3), then Julia will run a different compiled code which takes in two integers and returns the value 2a+b. The function f is an abstraction or a short-hand for the multitude of different methods which have the same form, and this design of using the symbol "f" to call all of these different methods is called multiple dispatch. And this goes all the way down: the + operator is actually a function which will call methods depending on the types it sees.

Julia actually gets its speed is because this compiled code knows its types, and so the compiled code that f(2.0,3.0) calls is exactly the compiled code that you would get by defining the same C/Fortran function which takes in floating point numbers. You can check this with the @code_native macro to see the compiled assembly:

```
@code_native f(2.0,3.0)

# This prints out the following:

pushq %rbp
movq %rsp, %rbp
Source line: 2
vaddsd %xmm0, %xmm0, %xmm0
vaddsd %xmm1, %xmm0, %xmm0
popq %rbp
retq
nop
```

This is the same compiled assembly you would expect from the C/Fortran function, and it is different than the assembly code for integers:

```
@code_native f(2,3)

pushq %rbp
movq %rsp, %rbp
Source line: 2
leaq (%rdx,%rcx,2), %rax
popq %rbp
retq
nopw (%rax,%rax)
```

THE MAIN POINT: THE REPL/GLOBAL SCOPE DOES NOT ALLOW TYPE SPECIFICITY

This brings us to the main point: The REPL / Global Scope is slow because it does not allow type specification. First of all, notice that the REPL is the global scope because Julia allows nested scoping for functions. For example, if we define

```
function outer()
  a = 5
  function inner()
    return 2a
  end
  b = inner()
  return 3a+b
end
```

you will see that this code works. This is because Julia allows you to grab the "a" from the outer function into the inner function. If you apply this idea recursively, then you understand the highest scope is the scope which is directly the REPL (which is the global scope of a module Main). But now let's think about how a function will compile in this situation. Let's do the same case as before, but using the globals:

```
a=2.0; a=3.0
function linearcombo()
  return 2a+b
end
ans = linearcombo()
a = 2; b = 3
ans2= linearcombo()
```

Question: What types should the compiler assume "a" and "b" are? Notice that in this example we changed the types and still called the same function. In order for this compiled C function to not segfault, it needs to be able to deal with whatever types we throw at it: floats, ints, arrays, weird user-defined types, etc. In Julia parlance, this means that the variables have to be "boxed", and the types are checked with every use. What do you think that compiled code looks like?

```
pushq %rbp
movq %rsp, %rbp
pushq %r15
pushq %r14
pushq %r12
pushq %rsi
pushq %rdi
pushq %rbx
subq $96, %rsp
movl $2147565792, %edi        # imm = 0x800140E0
movabsq $jl_get_ptls_states, %rax
callq *%rax
```

```
movq %rax, %rsi
leaq -72(%rbp), %r14
movq $0, -88(%rbp)
vxorps %xmm0, %xmm0, %xmm0
vmovups %xmm0, -72(%rbp)
movq $0, -56(%rbp)
movq $10, -104(%rbp)
movq (%rsi), %rax
movq %rax, -96(%rbp)
leaq -104(%rbp), %rax
movq %rax, (%rsi)
Source line: 3
movq pcre2_default_compile_context_8(%rdi), %rax
movq %rax, -56(%rbp)
movl $2154391480, %eax      # imm = 0x806967B8
vmovq %rax, %xmm0
vpslldq $8, %xmm0, %xmm0      # xmm0 = zero,zero,zero,zero,zero,zero,zero,zero,xmm0[0,1,2,3,4,5,6,7]
vmovdqu %xmm0, -80(%rbp)
movq %rdi, -64(%rbp)
movabsq $jl_apply_generic, %r15
movl $3, %edx
movq %r14, %rcx
callq *%r15
movq %rax, %rbx
movq %rbx, -88(%rbp)
movabsq $586874896, %r12      # imm = 0x22FB0010
movq (%r12), %rax
testq %rax, %rax
jne L198
leaq 98096(%rdi), %rcx
movabsq $jl_get_binding_or_error, %rax
movl $122868360, %edx      # imm = 0x752D288
callq *%rax
movq %rax, (%r12)
L198:
movq 8(%rax), %rax
testq %rax, %rax
je L263
movq %rax, -80(%rbp)
addq $5498232, %rdi      # imm = 0x53E578
movq %rdi, -72(%rbp)
movq %rbx, -64(%rbp)
movq %rax, -56(%rbp)
movl $3, %edx
movq %r14, %rcx
callq *%r15
movq -96(%rbp), %rcx
movq %rcx, (%rsi)
addq $96, %rsp
popq %rbx
popq %rdi
popq %rsi
popq %r12
popq %r14
popq %r15
popq %rbp
retq
L263:
movabsq $jl_undefined_var_error, %rax
movl $122868360, %ecx      # imm = 0x752D288
callq *%rax
ud2
nopw (%rax,%rax)
```

For dynamic languages without type-specialization, this bloated code with all of the extra instructions is as good as you can get, which is why Julia slows down to their speed.

To understand why this is a big deal, notice that every single piece of code that you write in Julia is compiled. So let's say you write a loop in your script:

```
a = 1
for i = 1:100
  a += a + f(a)
end
```

The compiler has to compile that loop, but since it cannot guarantee the types do not change, it conservatively gives that nasty long code, leading to slow execution.

HOW TO AVOID THE ISSUE

There are a few ways to avoid this issue. The simplest way is to always wrap your scripts in functions. For example, with the previous code we can do:

```
function geta(a)
  # can also just define a=1 here
  for i = 1:100
    a += a + f(a)
  end
  return a
end
a = geta(1)
```

This will give you the same output, but since the compiler is able to specialize on the type of a, it will give the performant compiled code that you want. Another thing you can do is define your variables as constants.

```
const b = 5
```

By doing this, you are telling the compiler that the variable will not change, and thus it will be able to specialize all of the code which uses it on the type that it currently is. There's a small quirk that Julia actually allows you to change the value of a constant, but not the type. Thus you can use "const" as a way to tell the compiler that you won't be changing the type and speed up your codes. However, note that there are some small quirks that come up since you guaranteed to the compiler the value won't change. For example:

```
const a = 5
f() = a
println(f()) # Prints 5
a = 6
println(f()) # Prints 5
```

this does not work as expected because the compiler, realizing that it knows the answer to "f()=a" (since a is a constant), simply replaced the function call with the answer, giving different behavior than if a was not a constant.

This is all just one big way of saying: **Don't write your scripts directly in the REPL, always wrap them in a function.**

Let's hit one related point as well.

GOTCHA #2: TYPE-INSTABILITIES

So I just made a huge point about how specializing code for the given types is crucial. Let me ask a quick question, what happens when your types can change?

If you guessed "well, you can't really specialize the compiled code in that case either", then you are correct. This kind of problem is known as a type-instability. These can show up in many different ways, but one common example is that you initialize a value in a way that is easy, but not necessarily that type that it should be. For example, let's look at:

```
function g()
  x=1
  for i = 1:10
    x = x/2
  end
  return x
end
```

Notice that "1/2" is a floating point number in Julia. Therefore it we started with "x=1", it will change types from an integer to a floating point number, and thus the function has to compile the inner loop as though it can be either type. If we instead had the function:

```
function h()
  x=1.0
  for i = 1:10
    x = x/2
```

```
  end
  return x
end
```

then the whole function can optimally compile knowing x will stay a floating point number (this ability for the compiler to judge types is known as type inference). We can check the compiled code to see the difference:

```
pushq %rbp
movq %rsp, %rbp
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rsi
pushq %rdi
pushq %rbx
subq $136, %rsp
movl $2147565728, %ebx      # imm = 0x800140A0
movabsq $jl_get_ptls_states, %rax
callq *%rax
movq %rax, -152(%rbp)
vxorps %xmm0, %xmm0, %xmm0
vmovups %xmm0, -80(%rbp)
movq $0, -64(%rbp)
vxorps %ymm0, %ymm0, %ymm0
vmovups %ymm0, -128(%rbp)
movq $0, -96(%rbp)
movq $18, -144(%rbp)
movq (%rax), %rcx
movq %rcx, -136(%rbp)
leaq -144(%rbp), %rcx
movq %rcx, (%rax)
movq $0, -88(%rbp)
Source line: 4
movq %rbx, -104(%rbp)
movl $10, %edi
leaq 477872(%rbx), %r13
leaq 10039728(%rbx), %r15
leaq 8958904(%rbx), %r14
leaq 64(%rbx), %r12
leaq 10126032(%rbx), %rax
movq %rax, -160(%rbp)
nopw (%rax,%rax)
L176:
movq %rbx, -128(%rbp)
movq -8(%rbx), %rax
andq $-16, %rax
movq %r15, %rcx
cmpq %r13, %rax
je L272
movq %rbx, -96(%rbp)
movq -160(%rbp), %rcx
cmpq $2147419568, %rax      # imm = 0x7FFF05B0
je L272
movq %rbx, -72(%rbp)
movq %r14, -80(%rbp)
movq %r12, -64(%rbp)
movl $3, %edx
leaq -80(%rbp), %rcx
movabsq $jl_apply_generic, %rax
vzeroupper
callq *%rax
movq %rax, -88(%rbp)
jmp L317
nopw %cs:(%rax,%rax)
L272:
movq %rcx, -120(%rbp)
movq %rbx, -72(%rbp)
```

```
movq %r14, -80(%rbp)
movq %r12, -64(%rbp)
movl $3, %r8d
leaq -80(%rbp), %rdx
movabsq $jl_invoke, %rax
vzeroupper
callq *%rax
movq %rax, -112(%rbp)
L317:
movq (%rax), %rsi
movl $1488, %edx            # imm = 0x5D0
movl $16, %r8d
movq -152(%rbp), %rcx
movabsq $jl_gc_pool_alloc, %rax
callq *%rax
movq %rax, %rbx
movq %r13, -8(%rbx)
movq %rsi, (%rbx)
movq %rbx, -104(%rbp)
Source line: 3
addq $-1, %rdi
jne L176
Source line: 6
movq -136(%rbp), %rax
movq -152(%rbp), %rcx
movq %rax, (%rcx)
movq %rbx, %rax
addq $136, %rsp
popq %rbx
popq %rdi
popq %rsi
popq %r12
popq %r13
popq %r14
popq %r15
popq %rbp
retq
nop
```

Verses:

```
pushq %rbp
movq %rsp, %rbp
movabsq $567811336, %rax        # imm = 0x21D81D08
Source line: 6
vmovsd (%rax), %xmm0            # xmm0 = mem[0],zero
popq %rbp
retq
nopw %cs:(%rax,%rax)
```

Notice how many fewer computational steps are required to compute the same value!

HOW TO FIND AND DEAL WITH TYPE-INSTABILITIES

At this point you might ask, "well, why not just use C so you don't have to try and find these instabilities?" The answer is:

1. They are easy to find
2. They can be useful
3. You can handle necessary instabilities with function barriers

HOW TO FIND TYPE-INSTABILITIES

Julia gives you the macro @code_warntype to show you where type instabilities are. For example, if we use this on the "g" function we created:

```
@code_warntype g()

Variables:
  #self#::#g
  x::ANY
  #temp#@_3::Int64
  i::Int64
```

```
  #temp#@_5::Core.MethodInstance
  #temp#@_6::Float64

Body:
  begin
      x::ANY = 1 # line 3:
      SSAValue(2) = (Base.select_value)((Base.sle_int)(1,10)::Bool,10,(Base.box)(Int64,(Base.sub_int)(1,1)))::Int64
      #temp#@_3::Int64 = 1
      5:
      unless (Base.box)(Base.Bool,(Base.not_int)((#temp#@_3::Int64 === (Base.box)(Int64,(Base.add_int)(SSAValue(2),1)))::Bool)) goto 30
      SSAValue(3) = #temp#@_3::Int64
      SSAValue(4) = (Base.box)(Int64,(Base.add_int)(#temp#@_3::Int64,1))
      i::Int64 = SSAValue(3)
      #temp#@_3::Int64 = SSAValue(4) # line 4:
      unless (Core.isa)(x::UNION{FLOAT64,INT64},Float64)::ANY goto 15
      #temp#@_5::Core.MethodInstance = MethodInstance for /(::Float64, ::Int64)
      goto 24
      15:
      unless (Core.isa)(x::UNION{FLOAT64,INT64},Int64)::ANY goto 19
      #temp#@_5::Core.MethodInstance = MethodInstance for /(::Int64, ::Int64)
      goto 24
      19:
      goto 21
      21:
      #temp#@_6::Float64 = (x::UNION{FLOAT64,INT64} / 2)::Float64
      goto 26
      24:
      #temp#@_6::Float64 = $(Expr(:invoke, :(#temp#@_5), :(Main./), :(x::Union{Float64,Int64}), 2))
      26:
      x::ANY = #temp#@_6::Float64
      28:
      goto 5
      30:  # line 6:
      return x::UNION{FLOAT64,INT64}
  end::UNION{FLOAT64,INT64}
```

Notice that it tells us at the top that the type of x is "ANY". It will capitalize any type which is not inferred as a "strict type", i.e. it is an abstract type which needs to be boxed/checked at each step. We see that at the end we return x as a "UNION{FLOAT64,INT64}", which is another non-strict type. This tells us that the type of x changed, causing the difficulty. If we instead look at the @code_warntype for h, we get all strict types:

```
@code_warntype h()

Variables:
  #self#::#h
  x::Float64
  #temp#::Int64
  i::Int64

Body:
  begin
      x::Float64 = 1.0 # line 3:
      SSAValue(2) = (Base.select_value)((Base.sle_int)(1,10)::Bool,10,(Base.box)(Int64,(Base.sub_int)(1,1)))::Int64
      #temp#::Int64 = 1
      5:
      unless (Base.box)(Base.Bool,(Base.not_int)((#temp#::Int64 === (Base.box)(Int64,(Base.add_int)(SSAValue(2),1)))::Bool)) goto 15
      SSAValue(3) = #temp#::Int64
      SSAValue(4) = (Base.box)(Int64,(Base.add_int)(#temp#::Int64,1))
      i::Int64 = SSAValue(3)
      #temp#::Int64 = SSAValue(4) # line 4:
      x::Float64 = (Base.box)(Base.Float64,(Base.div_float)(x::Float64,(Base.box)(Float64,(Base.sitofp)(Float64,2))))
      13:
      goto 5
      15:  # line 6:
      return x::Float64
  end::Float64
```

Indicating that this function is type stable and will compile to essentially optimal C code. Thus type-instabilities are not hard to find. What's harder is to find the right design.

WHY ALLOW TYPE-INSTABILITIES?

This is an age old question which has lead to dynamically-typed languages dominating the scripting language playing field. The idea is that, in many cases you want to make a tradeoff between performance and robustness. For example, you may want to read a table from a webpage which has numbers all mixed together with integers and floating point numbers. In Julia, you can write your function such that if they were all integers, it will compile well, and if they were all floating point numbers, it will also compile well. And if they're mixed? It will still work. That's the flexibility/convenience we know and love from a language like Python/R. But Julia will explicitly tell you (via @code_warntype) when you are making this performance tradeoff.

HOW TO HANDLE TYPE-INSTABILITIES

There are a few ways to handle type-instabilities. First of all, if you like something like C/Fortran where your types are declared and can't change (thus ensuring type-stability), you can do that in Julia. You can declare your types in a function with the following syntax:

```
local a::Int64 = 5
```

This makes "a" an 64-bit integer, and if future code tries to change it, an error will be thrown (or a proper conversion will be done. But since the conversion will not automatically round, it will most likely throw errors). Sprinkle these around your code and you will get type stability the C/Fortran way.

A less heavy handed way to handle this is with type-assertions. This is where you put the same syntax on the other side of the equals sign. For example:

```
a = (b/c)::Float64
```

This says "calculate b/c, and make sure that the output is a Float64. If it's not, try to do an auto-conversion. If it can't easily convert, throw an error". Putting these around will help you make sure you know the types which are involved.

However, there are cases where type instabilities are necessary. For example, let's say you want to have a robust code, but the user gives you something crazy like:

```
arr = Vector{Union{Int64,Float64},2}(4)
arr[1]=4
arr[2]=2.0
arr[3]=3.2
arr[4]=1
```

which is a 4x4 array of both integers and floating point numbers. The actual element type for the array is "Union{Int64,Float64}" which we saw before was a non-strict type which can lead to issues. The compiler only knows that each value can be either an integer or a floating point number, but not which element is which type. This means that naively performing arithmetic on this array, like:

```
function foo{T,N}(array::Array{T,N})
  for i in eachindex(array)
    val = array[i]
    # do algorithm X on val
  end
end
```

will be slow since the operations will be boxed.

However, we can use multiple-dispatch to run the codes in a type-specialized manner. This is known as using function barriers. For example:

```
function inner_foo{T
```

Notice that because of multiple-dispatch, calling inner_foo either calls a method specifically compiled for floating point numbers, or a methoc

Thus I hope you see that Julia offers a good mixture between the performance of strict typing and the convenience of dynamic typing. A goc

GOTCHA #3: EVAL RUNS AT THE GLOBAL SCOPE

One last typing issue: eval. Remember this: eval runs at the global scope.

One of the greatest strengths of Julia is its metaprogramming capabilities. This allows you to effortlessly write code which generates code, e

```
macro defa()
  :(a=5)
```

```
end
```

will replace any instance of "@defa" with the code "a=5" (":(a=5)" is the quoted expression for "a=5". Julia code is all expressions, and thus

However, sometimes you may need to directly evaluate the generated code. Julia gives you the "eval" function or the "@eval" macro for doi

```
@eval :(a=5)
```

then this will evaluate at the global scope (the REPL). Thus all of the associated problems will occur. However, the fix is the same as the fix

```
function testeval()
  @eval :(a=5)
  return 2a+5
end
```

will not give a good compiled code since "a" was essentially declared at the REPL. But we can use the tools from before to fix this. For exan

```
function testeval()
  @eval :(a=5)
  b = a::Int64
  return 2b+5
end
```

Here "b" is a local variable, and the compiler can infer that its type won't change and thus we have type-stability and are living in good perfo

That's the last of the gotcha's related to type-instability. You can see that there's a very common thread for why it occurs and how to handle

GOTCHA #4: HOW EXPRESSIONS BREAK UP

This is one that got me for awhile at first. In Julia, there are many cases where expressions will continue if they are not finished. For this rea

Easy rule, right? Just make sure you remember how functions finish. For example:

```
a = 2 + 3 + 4 + 5 + 6 + 7
   +8 + 9 + 10+ 11+ 12+ 13
```

looks like it will evaluate to 90, but instead it gives 27. Why? Because "a = 2 + 3 + 4 + 5 + 6 + 7" is a complete expression, so it will make "a

```
a = 2 + 3 + 4 + 5 + 6 + 7 +
    8 + 9 + 10+ 11+ 12+ 13
```

This will make a=90 as we wanted. This might trip you up the first time, but then you'll get used to it.

The more difficult issue dealing with array definitions. For example:

```
x = rand(2,2)
a = [cos(2*pi.*x[:,1]).*cos(2*pi.*x[:,2])./(4*pi) -sin(2.*x[:,1]).*sin(2.*x[:,2])./(4)]
b = [cos(2*pi.*x[:,1]).*cos(2*pi.*x[:,2])./(4*pi) - sin(2.*x[:,1]).*sin(2.*x[:,2])./(4)]
```

at glance you might think a and b are the same, but they are not! The first will give you a (2,2) matrix, while the second is a (1-dimensional)

```
a = [1 -2]
b = [1 - 2]
```

In the first case there are two numbers: "1" and "-2". In the second there is an expression: "1-2" (which is evaluated to give the array [-1]). Tl

```
a = [1 2 3 -4
     2 -3 1 4]
```

and get the 2x4 matrix that you'd expect. However, this is the tradeoff that occurs. However, this issue is also easy to avoid: instead of conc

```
a = hcat(cos(2*pi.*x[:,1]).*cos(2*pi.*x[:,2])./(4*pi),-sin(2.*x[:,1]).*sin(2.*x[:,2])./(4))
```

Problem solved!

GOTCHA #5: VIEWS, COPY, AND DEEPCOPY

One way in which Julia gets good performance is by working with "views". An "Array" is actually a "view" to the contiguous blot of memory w

```
a = [3;4;5]
b = a
b[1] = 1
```

then at the end we will have that "a" is the array "[1;4;5]", i.e. changing "b" changes "a". The reason is "b=a" set the value of "b" to the value

This is very useful because it also allows you to keep the same array in many different forms. For example, we can have both a matrix and t

```
a = rand(2,2) # Makes a random 2x2 matrix
b = vec(a) # Makes a view to the 2x2 matrix which is a 1-dimensional array
```

Now "b" is a vector, but changing "b" still changes "a", where "b" is indexed by reading down the columns. Notice that this whole time, no ar

Now some details. Notice that the syntax for slicing an array will create a copy when on the right-hand side. For example:

```
c = a[1:2,1]
```

will create a new array, and point "c" to that new array (thus changing "c" won't change "a"). This can be necessary behavior, however note

```
d = @view a[1:2,1]
e = view(a,1:2,1)
```

Both "d" and "e" are the same thing, and changing either "d" or "e" will change "a" because both will not copy the array, just make a new var

If this syntax is on the left-hand side, then it's a view. For example:

```
a[1:2,1] = [1;2]
```

will change "a" because, on the left-hand side, "a[1:2,1]" is the same as "view(a,1:2,1)" which points to the same memory as "a".

What if we need to make copies? Then we can use the copy function:

```
b = copy(a)
```

Now since "b" is a copy of "a" and not a view, changing "b" will not change "a". If we had already defined "a", there's a handy in-place copy '

But now let's make a slightly more complicated array. For example, let's make a "Vector{Vector}":

```
a = Vector{Vector{Float64}}(2)
a[1] = [1;2;3]
a[2] = [4;5;6]
```

Each element of "a" is a vector. What happens when we copy a?

```
b = copy(a)
b[1][1] = 10
```

Notice that this will change a[1][1] to 10 as well! Why did this happen? What happened is we used "copy" to copy the values of "a". But the v

```
b = deepcopy(a)
```

This recursively calls copy in such a manner that we avoid this issue. Again, the rules of Julia are very simple and there's no magic, but som

GOTCHA #6: TEMPORARY ALLOCATIONS, VECTORIZATION, AND IN-PLACE FUNCTIONS

In MATLAB/Python/R, you're told to use vectorization. In Julia you might have heard that "devectorized code is better". I wrote about this pa

For this reason, you will want to fuse your vectorized operations and write them in-place in order to avoid allocations. What do I mean by in-

```
function f()
  x = [1;5;6]
  for i = 1:10
    x = x + inner(x)
  end
  return x
end
function inner(x)
  return 2x
end
```

then each time inner is called, it will create a new array to return "2x" in. Clearly we don't need to keep making new arrays. So instead we co

```
function f()
  x = [1;5;6]
  y = Vector{Int64}(3)
  for i = 1:10
    inner(y,x)
    for i in 1:3
      x[i] = x[i] + y[i]
    end
    copy!(y,x)
  end
  return x
end
function inner!(y,x)
  for i=1:3
    y[i] = 2*x[i]
  end
  nothing
end
```

Let's dig into what's happening here. "inner!(y,x)" doesn't return anything, but it changes "y". Since "y" is an array, the value of "y" is the poir

In the same way, "copy!(y,x)" is an in-place function which writes the values of "x" to "y", updating it. As you can see, this means that every

It's nice that we can get fast, but the syntax bloated a little when we had to write out the loops. That's where loop-fusion comes in. In Julia v0

```
x .= x .+ f.(x)
```

The ".=" will do element-wise equals, so this will essentially turn be the code

```
for i = 1:length(x)
  x[i] = x[i] + f(x[i])
end
```

which is the allocation-free loop we wanted! Thus another way to write our function would've been:

```
function f()
  x = [1;5;6]
  for i = 1:10
    x .= x .+ inner.(x)
  end
  return x
end
function inner(x)
  return 2x
end
```

Therefore we still get the concise vectorized syntax of MATLAB/R/Python, but this version doesn't create temporary arrays and thus will be

**** Note: Some operators do not fuse in v0.5. For example, ".*" won't fuse yet. This is still a work in progress but should be all together by v

GOTCHA #7: NOT BUILDING THE SYSTEM IMAGE FOR YOUR HARDWARE

This is actually something I fell prey to for a very long time. I was following all of these rules thinking I was a Julia champ, and then one day

It turns out that the pre-built binaries that you get via the downloads off the Julia site are toned-down in their capabilities in order to be usabl

Luckily there's an easy fix provided by Mustafa Mohamad (@musm). Just run the following code in Julia:

```
include(joinpath(dirname(JULIA_HOME),"share","julia","build_sysimg.jl")); build_sysimg(force=true)
```

If you're on Windows, you may need to run this code first:

```
Pkg.add("WinRPM");
WinRPM.install("gcc", yes=true)
WinRPM.install("winpthreads-devel", yes=true)
```

And on any system, you may need to have administrator privileges. This will take a little bit but when it's done, your install will be tuned to yo

CONCLUSION: LEARN THE RULES, UNDERSTAND THEM, THEN PROFIT

To reiterate one last time: Julia doesn't have compiler magic, just simple rules. Learn the rules well and all of this will be second nature. I ho

Here's a question for you: what Julia gotchas did I miss? Leave a comment explaining a gotcha and how to handle it. Also, just for fun, what