The Need for Speed

Jesse Perla, Thomas J. Sargent and John Stachurski

December 4, 2020

1 Contents

- Overview 2
- Understanding Multiple Dispatch in Julia 3
- Foundations 4
- JIT Compilation in Julia 5
- Fast and Slow Julia Code 6
- Further Comments 7

2 Overview

Computer scientists often classify programming languages according to the following two categories.

High level languages aim to maximize productivity by

- being easy to read, write and debug
- automating standard tasks (e.g., memory management)
- being interactive, etc.

Low level languages aim for speed and control, which they achieve by

- being closer to the metal (direct access to CPU, memory, etc.)
- requiring a relatively large amount of information from the user (e.g., all data types must be specified)

Traditionally we understand this as a trade off

- high productivity or high performance
- optimized for humans or optimized for machines

One of the great strengths of Julia is that it pushes out the curve, achieving both high productivity and high performance with relatively little fuss.

The word “relatively” is important here, however…

In simple programs, excellent performance is often trivial to achieve.

For longer, more sophisticated programs, you need to be aware of potential stumbling blocks.

This lecture covers the key points.

2.1 Requirements

You should read our earlier lecture on types, methods and multiple dispatch before this one.

2.2 Setup

```
In [1]: using InstantiateFromURL
        # optionally add arguments to force installation: instantiate = true,
 ↪precompile = true
        github_project("QuantEcon/quantecon-notebooks-julia", version = "0.8.0")
```

```
In [2]: using LinearAlgebra, Statistics
```

3 Understanding Multiple Dispatch in Julia

This section provides more background on how methods, functions, and types are connected.

3.1 Methods and Functions

The precise data type is important, for reasons of both efficiency and mathematical correctness.

For example consider 1 + 1 vs. 1.0 + 1.0 or [1 0] + [0 1].

On a CPU, integer and floating point addition are different things, using a different set of instructions.

Julia handles this problem by storing multiple, specialized versions of functions like addition, one for each data type or set of data types.

These individual specialized versions are called **methods**.

When an operation like addition is requested, the Julia compiler inspects the type of data to be acted on and hands it out to the appropriate method.

This process is called **multiple dispatch**.

Like all “infix” operators, 1 + 1 has the alternative syntax +(1, 1)

```
In [3]: +(1, 1)
```

```
Out[3]: 2
```

This operator + is itself a function with multiple methods.

We can investigate them using the @which macro, which shows the method to which a given call is dispatched

```
In [4]: x, y = 1.0, 1.0
        @which +(x, y)
```

```
Out[4]: +(x::Float64, y::Float64) in Base at float.jl:401
```

We see that the operation is sent to the `+` method that specializes in adding floating point numbers.

Here's the integer case

```
In [5]: x, y = 1, 1
        @which +(x, y)
```

```
Out[5]: +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128,
  ↪UInt16, UInt32,
        UInt64, UInt8} in Base at int.jl:53
```

This output says that the call has been dispatched to the + method responsible for handling integer values.

(We'll learn more about the details of this syntax below)

Here's another example, with complex numbers

```
In [6]: x, y = 1.0 + 1.0im, 1.0 + 1.0im
        @which +(x, y)
```

```
Out[6]: +(z::Complex, w::Complex) in Base at complex.jl:275
```

Again, the call has been dispatched to a + method specifically designed for handling the given data type.

3.1.1 Adding Methods

It's straightforward to add methods to existing functions.

For example, we can't at present add an integer and a string in Julia (i.e. `100 + "100"` is not valid syntax).

This is sensible behavior, but if you want to change it there's nothing to stop you.

```
In [7]: import Base: +  # enables adding methods to the + function

        +(x::Integer, y::String) = x + parse(Int, y)

        @show +(100, "100")
        @show 100 + "100";  # equivalent
```

```
    100 + "100" = 200
100 + "100" = 200
```

3.2 Understanding the Compilation Process

We can now be a little bit clearer about what happens when you call a function on given types.

Suppose we execute the function call `f(a, b)` where `a` and `b` are of concrete types `S` and `T` respectively.

The Julia interpreter first queries the types of `a` and `b` to obtain the tuple `(S, T)`.

It then parses the list of methods belonging to `f`, searching for a match.

If it finds a method matching `(S, T)` it calls that method.

If not, it looks to see whether the pair `(S, T)` matches any method defined for *immediate parent types*.

For example, if `S` is `Float64` and `T` is `ComplexF32` then the immediate parents are `AbstractFloat` and `Number` respectively

```
In [8]: supertype(Float64)
```

```
Out[8]: AbstractFloat
```

```
In [9]: supertype(ComplexF32)
```

```
Out[9]: Number
```

Hence the interpreter looks next for a method of the form `f(x::AbstractFloat, y::Number)`.

If the interpreter can't find a match in immediate parents (supertypes) it proceeds up the tree, looking at the parents of the last type it checked at each iteration.

- If it eventually finds a matching method, it invokes that method.
- If not, we get an error.

This is the process that leads to the following error (since we only added the `+` for adding `Integer` and `String` above)

```
In [10]: @show (typeof(100.0) <: Integer) == false
         100.0 + "100"
```

```
(typeof(100.0) <: Integer) == false = true
```

```
    MethodError: no method matching +(::Float64, ::String)
  Closest candidates are:
    +(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:
↪529
    +(::Float64, !Matched::Float64) at float.jl:401
    +(::AbstractFloat, !Matched::Bool) at bool.jl:106
    ...
```

```
Stacktrace:

 [1] top-level scope at In[10]:2
```

Because the dispatch procedure starts from concrete types and works upwards, dispatch always invokes the *most specific method* available.

For example, if you have methods for function `f` that handle

1. `(Float64, Int64)` pairs
2. `(Number, Number)` pairs

and you call `f` with `f(0.5, 1)` then the first method will be invoked.

This makes sense because (hopefully) the first method is optimized for exactly this kind of data.

The second method is probably more of a “catch all” method that handles other data in a less optimal way.

Here’s another simple example, involving a user-defined function

```
In [11]: function q(x)  # or q(x::Any)
             println("Default (Any) method invoked")
         end

         function q(x::Number)
             println("Number method invoked")
         end

         function q(x::Integer)
             println("Integer method invoked")
         end
```

```
Out[11]: q (generic function with 3 methods)
```

Let’s now run this and see how it relates to our discussion of method dispatch above

```
In [12]: q(3)
```

```
Integer method invoked
```

```
In [13]: q(3.0)
```

```
Number method invoked
```

```
In [14]: q("foo")
```

```
    Default (Any) method invoked
```

Since `typeof(3) <: Int64 <: Integer <: Number`, the call `q(3)` proceeds up the tree to `Integer` and invokes `q(x::Integer)`.

On the other hand, `3.0` is a `Float64`, which is not a subtype of `Integer`.

Hence the call `q(3.0)` continues up to `q(x::Number)`.

Finally, `q("foo")` is handled by the function operating on `Any`, since `String` is not a subtype of `Number` or `Integer`.

3.3 Analyzing Function Return Types

For the most part, time spent "optimizing" Julia code to run faster is about ensuring the compiler can correctly deduce types for all functions.

The macro `@code_warntype` gives us a hint

```
In [15]: x = [1, 2, 3]
         f(x) = 2x
         @code_warntype f(x)
```

```
    Variables
  #self#::Core.Compiler.Const(f, false)
  x::Array{Int64,1}

Body::Array{Int64,1}
1 ─ %1 = (2 * x)::Array{Int64,1}
└──      return %1
```

The `@code_warntype` macro compiles `f(x)` using the type of `x` as an example – i.e., the `[1, 2, 3]` is used as a prototype for analyzing the compilation, rather than simply calculating the value.

Here, the `Body::Array{Int64,1}` tells us the type of the return value of the function, when called with types like `[1, 2, 3]`, is always a vector of integers.

In contrast, consider a function potentially returning `nothing`, as in this lecture

```
In [16]: f(x) = x > 0.0 ? x : nothing
         @code_warntype f(1)
```

```
    Variables
  #self#::Core.Compiler.Const(f, false)
  x::Int64

Body::Union{Nothing, Int64}
1 ─ %1 = (x > 0.0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return Main.nothing
```

This states that the compiler determines the return type when called with an integer (like `1`) could be one of two different types, `Body::Union{Nothing, Int64}`.

A final example is a variation on the above, which returns the maximum of `x` and `0`.

In [17]:
```
f(x) = x > 0.0 ? x : 0.0
@code_warntype f(1)
```

```
  Variables
#self#::Core.Compiler.Const(f, false)
x::Int64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0.0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0.0
```

Which shows that, when called with an integer, the type could be that integer or the floating point `0.0`.

On the other hand, if we use change the function to return `0` if x <= 0, it is type-unstable with floating point.

In [18]:
```
f(x) = x > 0.0 ? x : 0
@code_warntype f(1.0)
```

```
  Variables
#self#::Core.Compiler.Const(f, false)
x::Float64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0.0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─      return 0
```

The solution is to use the `zero(x)` function which returns the additive identity element of type `x`.

On the other hand, if we change the function to return `0` if `x <= 0`, it is type-unstable with floating point.

In [19]:
```
@show zero(2.3)
@show zero(4)
@show zero(2.0 + 3im)

f(x) = x > 0.0 ? x : zero(x)
@code_warntype f(1.0)
```

```
  zero(2.3) = 0.0
zero(4) = 0
zero(2.0 + 3im) = 0.0 + 0.0im
Variables
```

```
  #self#::Core.Compiler.Const(f, false)
  x::Float64

Body::Float64
1 ─ %1 = (x > 0.0)::Bool
└──      goto #3 if not %1
2 ─      return x
3 ─ %4 = Main.zero(x)::Core.Compiler.Const(0.0, false)
└──      return %4
```

4 Foundations

Let's think about how quickly code runs, taking as given

- hardware configuration
- algorithm (i.e., set of instructions to be executed)

We'll start by discussing the kinds of instructions that machines understand.

4.1 Machine Code

All instructions for computers end up as *machine code*.

Writing fast code — expressing a given algorithm so that it runs quickly — boils down to producing efficient machine code.

You can do this yourself, by hand, if you want to.

Typically this is done by writing assembly, which is a symbolic representation of machine code.

Here's some assembly code implementing a function that takes arguments a, b and returns $2a + 8b$

```
pushq   %rbp
movq    %rsp, %rbp
addq    %rdi, %rdi
leaq    (%rdi,%rsi,8), %rax
popq    %rbp
retq
nopl    (%rax)
```

Note that this code is specific to one particular piece of hardware that we use — different machines require different machine code.

If you ever feel tempted to start rewriting your economic model in assembly, please restrain yourself.

It's far more sensible to give these instructions in a language like Julia, where they can be easily written and understood.

```
In [20]: function f(a, b)
             y = 2a + 8b
             return y
         end
```

```
Out[20]: f (generic function with 2 methods)
```

or Python

```python
def f(a, b):
    y = 2 * a + 8 * b
    return y
```

or even C

```c
int f(int a, int b) {
    int y = 2 * a + 8 * b;
    return y;
}
```

In any of these languages we end up with code that is much easier for humans to write, read, share and debug.

We leave it up to the machine itself to turn our code into machine code.

How exactly does this happen?

4.2 Generating Machine Code

The process for turning high level code into machine code differs across languages.

Let's look at some of the options and how they differ from one another.

4.2.1 AOT Compiled Languages

Traditional compiled languages like Fortran, C and C++ are a reasonable option for writing fast code.

Indeed, the standard benchmark for performance is still well-written C or Fortran.

These languages compile down to efficient machine code because users are forced to provide a lot of detail on data types and how the code will execute.

The compiler therefore has ample information for building the corresponding machine code ahead of time (AOT) in a way that

- organizes the data optimally in memory and
- implements efficient operations as required for the task in hand

At the same time, the syntax and semantics of C and Fortran are verbose and unwieldy when compared to something like Julia.

Moreover, these low level languages lack the interactivity that's so crucial for scientific work.

4.2.2 Interpreted Languages

Interpreted languages like Python generate machine code "on the fly", during program execution.

This allows them to be flexible and interactive.

Moreover, programmers can leave many tedious details to the runtime environment, such as

- specifying variable types
- memory allocation/deallocation, etc.

But all this convenience and flexibility comes at a cost: it's hard to turn instructions written in these languages into efficient machine code.

For example, consider what happens when Python adds a long list of numbers together.

Typically the runtime environment has to check the type of these objects one by one before it figures out how to add them.

This involves substantial overheads.

There are also significant overheads associated with accessing the data values themselves, which might not be stored contiguously in memory.

The resulting machine code is often complex and slow.

4.2.3 Just-in-time compilation

Just-in-time (JIT) compilation is an alternative approach that marries some of the advantages of AOT compilation and interpreted languages.

The basic idea is that functions for specific tasks are compiled as requested.

As long as the compiler has enough information about what the function does, it can in principle generate efficient machine code.

In some instances, all the information is supplied by the programmer.

In other cases, the compiler will attempt to infer missing information on the fly based on usage.

Through this approach, computing environments built around JIT compilers aim to

- provide all the benefits of high level languages discussed above and, at the same time,
- produce efficient instruction sets when functions are compiled down to machine code

5 JIT Compilation in Julia

JIT compilation is the approach used by Julia.

In an ideal setting, all information necessary to generate efficient native machine code is supplied or inferred.

In such a setting, Julia will be on par with machine code from low level languages.

5.1 An Example

Consider the function

```
In [21]: function f(a, b)
             y = (a + 8b)^2
             return 7y
         end
```

```
Out[21]: f (generic function with 2 methods)
```

Suppose we call `f` with integer arguments (e.g., `z = f(1, 2)`).

The JIT compiler now knows the types of `a` and `b`.

Moreover, it can infer types for other variables inside the function

- e.g., `y` will also be an integer

It then compiles a specialized version of the function to handle integers and stores it in memory.

We can view the corresponding machine code using the @code_native macro

```
In [22]: @code_native f(1, 2)
```

```
            .text
; ┌ @ In[21]:2 within `f'
; │┌ @ In[21]:2 within `+'
        leaq    (%rdi,%rsi,8), %rcx
; │└
; │┌ @ intfuncs.jl:261 within `literal_pow'
; ││┌ @ int.jl:54 within `*'
        imulq   %rcx, %rcx
; │└└
; │ @ In[21]:3 within `f'
; │┌ @ int.jl:54 within `*'
        leaq    (,%rcx,8), %rax
        subq    %rcx, %rax
; │└
        retq
        nopw    %cs:(%rax,%rax)
        nop
; └
```

If we now call `f` again, but this time with floating point arguments, the JIT compiler will once more infer types for the other variables inside the function.

- e.g., `y` will also be a float

It then compiles a new version to handle this type of argument.

```
In [23]: @code_native f(1.0, 2.0)
```

```
	.text
; ┌ @ In[21]:2 within `f'
	movabsq $139778809481240, %rax  # imm = 0x7F20CA494818
; │┌ @ promotion.jl:312 within `*' @ float.jl:405
	vmulsd	(%rax), %xmm1, %xmm1
; │└
; │┌ @ float.jl:401 within `+'
	vaddsd	%xmm0, %xmm1, %xmm0
; │└
; │┌ @ intfuncs.jl:261 within `literal_pow'
; ││┌ @ float.jl:405 within `*'
	vmulsd	%xmm0, %xmm0, %xmm0
	movabsq $139778809481248, %rax  # imm = 0x7F20CA494820
; │└└
; │ @ In[21]:3 within `f'
; │┌ @ promotion.jl:312 within `*' @ float.jl:405
	vmulsd	(%rax), %xmm0, %xmm0
; │└
	retq
	nopw	%cs:(%rax,%rax)
	nop
; └
```

Subsequent calls using either floats or integers are now routed to the appropriate compiled code.

5.2 Potential Problems

In some senses, what we saw above was a best case scenario.

Sometimes the JIT compiler produces messy, slow machine code.

This happens when type inference fails or the compiler has insufficient information to optimize effectively.

The next section looks at situations where these problems arise and how to get around them.

6 Fast and Slow Julia Code

To summarize what we've learned so far, Julia provides a platform for generating highly efficient machine code with relatively little effort by combining

1. JIT compilation
2. Optional type declarations and type inference to pin down the types of variables and hence compile efficient code
3. Multiple dispatch to facilitate specialization and optimization of compiled code for different data types

But the process is not flawless, and hiccups can occur.

The purpose of this section is to highlight potential issues and show you how to circumvent them.

6.1 BenchmarkTools

The main Julia package for benchmarking is BenchmarkTools.jl.

Below, we'll use the `@btime` macro it exports to evaluate the performance of Julia code.

As mentioned in an earlier lecture, we can also save benchmark results to a file and guard against performance regressions in code.

For more, see the package docs.

6.2 Global Variables

Global variables are names assigned to values outside of any function or type definition.

The are convenient and novice programmers typically use them with abandon.

But global variables are also dangerous, especially in medium to large size programs, since

- they can affect what happens in any part of your program
- they can be changed by any function

This makes it much harder to be certain about what some small part of a given piece of code actually commands.

Here's a useful discussion on the topic.

When it comes to JIT compilation, global variables create further problems.

The reason is that the compiler can never be sure of the type of the global variable, or even that the type will stay constant while a given function runs.

To illustrate, consider this code, where `b` is global

```
In [24]: b = 1.0
         function g(a)
             global b
             for i ∈ 1:1_000_000
                 tmp = a + b
             end
         end
```

```
Out[24]: g (generic function with 1 method)
```

The code executes relatively slowly and uses a huge amount of memory.

```
In [25]: using BenchmarkTools

         @btime g(1.0)
```

```
    30.018 ms (2000000 allocations: 30.52 MiB)
```

If you look at the corresponding machine code you will see that it's a mess.

```
In [26]: @code_native g(1.0)
```

```
            .text
; ┌ @ In[24]:3 within `g'
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %r15
        pushq   %r14
        pushq   %r13
        pushq   %r12
        pushq   %rbx
        andq    $-32, %rsp
        subq    $128, %rsp
        vmovsd  %xmm0, 24(%rsp)
        vxorps  %xmm0, %xmm0, %xmm0
        vmovaps %ymm0, 32(%rsp)
        movq    %fs:0, %rax
        movq    $8, 32(%rsp)
        movq    -15712(%rax), %rcx
        movq    %rcx, 40(%rsp)
        leaq    32(%rsp), %rcx
        movq    %rcx, -15712(%rax)
        leaq    -15712(%rax), %r12
        movl    $1000000, %ebx          # imm = 0xF4240
        movabsq $jl_system_image_data, %r14
        leaq    88(%rsp), %r15
        nopl    (%rax)
; │ @ In[24]:5 within `g'
L112:
        movabsq $139778580060280, %rax  # imm = 0x7F20BC9C9878
        movq    (%rax), %r13
        movq    %r13, 48(%rsp)
        movq    %r12, %rdi
        movl    $1400, %esi             # imm = 0x578
        movl    $16, %edx
        movabsq $jl_gc_pool_alloc, %rax
        vzeroupper
        callq   *%rax
        movabsq $jl_system_image_data, %rcx
        movq    %rcx, -8(%rax)
        vmovsd  24(%rsp), %xmm0         # xmm0 = mem[0],zero
        vmovsd  %xmm0, (%rax)
        movq    %rax, 56(%rsp)
        movq    %rax, 88(%rsp)
        movq    %r13, 96(%rsp)
        movq    %r14, %rdi
        movq    %r15, %rsi
        movl    $2, %edx
        movabsq $jl_apply_generic, %rax
        callq   *%rax
; │┌ @ range.jl:597 within `iterate'
; ││┌ @ promotion.jl:398 within `=='
        addq    $-1, %rbx
; │└└
        jne     L112
        movq    40(%rsp), %rax
        movq    %rax, (%r12)
; │ @ In[24]:5 within `g'
        leaq    -40(%rbp), %rsp
        popq    %rbx
        popq    %r12
        popq    %r13
        popq    %r14
        popq    %r15
        popq    %rbp
        retq
        nopw    (%rax,%rax)
; └
```

If we eliminate the global variable like so

```
In [27]: function g(a, b)
             for i ∈ 1:1_000_000
                 tmp = a + b
             end
         end
```

```
Out[27]: g (generic function with 2 methods)
```

then execution speed improves dramatically

```
In [28]: @btime g(1.0, 1.0)
```

```
      1.599 ns (0 allocations: 0 bytes)
```

Note that the second run was dramatically faster than the first.

That's because the first call included the time for JIT compilaiton.

Notice also how small the memory footprint of the execution is.

Also, the machine code is simple and clean

```
In [29]: @code_native g(1.0, 1.0)
```

```
            .text
; ┌ @ In[27]:2 within `g'
        retq
        nopw    %cs:(%rax,%rax)
        nopl    (%rax,%rax)
; └
```

Now the compiler is certain of types throughout execution of the function and hence can optimize accordingly.

6.2.1 The `const` keyword

Another way to stabilize the code above is to maintain the global variable but prepend it with `const`

```
In [30]: const b_const = 1.0
         function g(a)
             global b_const
             for i ∈ 1:1_000_000
                 tmp = a + b_const
             end
         end
```

```
Out[30]: g (generic function with 2 methods)
```

Now the compiler can again generate efficient machine code.

We'll leave you to experiment with it.

6.3 Composite Types with Abstract Field Types

Another scenario that trips up the JIT compiler is when composite types have fields with abstract types.

We met this issue earlier, when we discussed AR(1) models.

Let's experiment, using, respectively,

- an untyped field
- a field with abstract type, and
- parametric typing

As we'll see, the last of these options gives us the best performance, while still maintaining significant flexibility.

Here's the untyped case

```
In [31]: struct Foo_generic
             a
         end
```

Here's the case of an abstract type on the field `a`

```
In [32]: struct Foo_abstract
             a::Real
         end
```

Finally, here's the parametrically typed case

```
In [33]: struct Foo_concrete{T <: Real}
             a::T
         end
```

Now we generate instances

```
In [34]: fg = Foo_generic(1.0)
         fa = Foo_abstract(1.0)
         fc = Foo_concrete(1.0)

Out[34]: Foo_concrete{Float64}(1.0)
```

In the last case, concrete type information for the fields is embedded in the object

```
In [35]: typeof(fc)

Out[35]: Foo_concrete{Float64}
```

This is significant because such information is detected by the compiler.

6.3.1 Timing

Here’s a function that uses the field `a` of our objects

```
In [36]: function f(foo)
             for i ▯ 1:1_000_000
                 tmp = i + foo.a
             end
         end
```

```
Out[36]: f (generic function with 2 methods)
```

Let’s try timing our code, starting with the generic case:

```
In [37]: @btime f($fg)
```

```
     39.721 ms (1999489 allocations: 30.51 MiB)
```

The timing is not very impressive.

Here’s the nasty looking machine code

```
In [38]: @code_native f(fg)
```

```
            .text
; ┌ @ In[36]:2 within `f'
        pushq   %rbp
        pushq   %r15
        pushq   %r14
        pushq   %r13
        pushq   %r12
        pushq   %rbx
        subq    $56, %rsp
        vxorps  %xmm0, %xmm0, %xmm0
        vmovaps %xmm0, (%rsp)
        movq    $0, 16(%rsp)
        movq    %rsi, 48(%rsp)
        movq    %fs:0, %rax
        movq    $4, (%rsp)
        movq    -15712(%rax), %rcx
        movq    %rcx, 8(%rsp)
        movq    %rsp, %rcx
        movq    %rcx, -15712(%rax)
        leaq    -15712(%rax), %rax
        movq    %rax, 24(%rsp)
        movq    (%rsi), %r13
        movl    $1, %ebx
        movabsq $jl_apply_generic, %r12
        movabsq $jl_system_image_data, %r14
        leaq    32(%rsp), %r15
        nopl    (%rax)
; │ @ In[36]:3 within `f'
; │┌ @ Base.jl:33 within `getproperty'
L128:
        movq    (%r13), %rbp
; │└
```

```
        movq    %rbx, %rdi
        movabsq $jl_box_int64, %rax
        callq   *%rax
        movq    %rax, 16(%rsp)
        movq    %rax, 32(%rsp)
        movq    %rbp, 40(%rsp)
        movq    %r14, %rdi
        movq    %r15, %rsi
        movl    $2, %edx
        callq   *%r12
; │┌ @ range.jl:597 within `iterate'
        addq    $1, %rbx
; ││┌ @ promotion.jl:398 within `=='
        cmpq    $1000001, %rbx          # imm = 0xF4241
; │└└
        jne     L128
        movq    8(%rsp), %rax
        movq    24(%rsp), %rcx
        movq    %rax, (%rcx)
        movabsq $jl_system_image_data, %rax
; │ @ In[36]:3 within `f'
        addq    $56, %rsp
        popq    %rbx
        popq    %r12
        popq    %r13
        popq    %r14
        popq    %r15
        popq    %rbp
        retq
        nopw    %cs:(%rax,%rax)
        nopl    (%rax)
; └
```

The abstract case is similar

```
In [39]: @btime f($fa)
```

```
      39.609 ms (1999489 allocations: 30.51 MiB)
```

Note the large memory footprint.

The machine code is also long and complex, although we omit details.

Finally, let's look at the parametrically typed version

```
In [40]: @btime f($fc)
```

```
      1.599 ns (0 allocations: 0 bytes)
```

Some of this time is JIT compilation, and one more execution gets us down to.

Here's the corresponding machine code

```
In [41]: @code_native f(fc)
```

```
            .text
; ┌ @ In[36]:2 within `f'
        retq
        nopw    %cs:(%rax,%rax)
        nopl    (%rax,%rax)
; └
```

Much nicer…

6.4 Abstract Containers

Another way we can run into trouble is with abstract container types.

Consider the following function, which essentially does the same job as Julia's `sum()` function but acts only on floating point data

```
In [42]: function sum_float_array(x::AbstractVector{<:Number})
             sum = 0.0
             for i ∈ eachindex(x)
                 sum += x[i]
             end
             return sum
         end

Out[42]: sum_float_array (generic function with 1 method)
```

Calls to this function run very quickly

```
In [43]: x = range(0,  1, length = Int(1e6))
         x = collect(x)
         typeof(x)

Out[43]: Array{Float64,1}

In [44]: @btime sum_float_array($x)

           1.251 ms (0 allocations: 0 bytes)


Out[44]: 499999.9999999796
```

When Julia compiles this function, it knows that the data passed in as `x` will be an array of 64 bit floats.

Hence it's known to the compiler that the relevant method for `+` is always addition of floating point numbers.

Moreover, the data can be arranged into continuous 64 bit blocks of memory to simplify memory access.

Finally, data types are stable — for example, the local variable `sum` starts off as a float and remains a float throughout.

6.4.1 Type Inferences

Here's the same function minus the type annotation in the function signature

```
In [45]: function sum_array(x)
             sum = 0.0
             for i  eachindex(x)
                 sum += x[i]
             end
             return sum
         end
```

```
Out[45]: sum_array (generic function with 1 method)
```

When we run it with the same array of floating point numbers it executes at a similar speed as the function with type information.

```
In [46]: @btime sum_array($x)
```

```
  1.251 ms (0 allocations: 0 bytes)
```

```
Out[46]: 499999.9999999796
```

The reason is that when `sum_array()` is first called on a vector of a given data type, a newly compiled version of the function is produced to handle that type.

In this case, since we're calling the function on a vector of floats, we get a compiled version of the function with essentially the same internal representation as `sum_float_array()`.

6.4.2 An Abstract Container

Things get tougher for the interpreter when the data type within the array is imprecise.

For example, the following snippet creates an array where the element type is `Any`

```
In [47]: x = Any[ 1/i for i  1:1e6 ];
```

```
In [48]: eltype(x)
```

```
Out[48]: Any
```

Now summation is much slower and memory management is less efficient.

```
In [49]: @btime sum_array($x)
```

```
  27.232 ms (1000000 allocations: 15.26 MiB)
```

```
Out[49]: 14.392726722864989
```

7 Further Comments

Here are some final comments on performance.

7.1 Explicit Typing

Writing fast Julia code amounts to writing Julia from which the compiler can generate efficient machine code.

For this, Julia needs to know about the type of data it's processing as early as possible.

We could hard code the type of all variables and function arguments but this comes at a cost.

Our code becomes more cumbersome and less generic.

We are starting to loose the advantages that drew us to Julia in the first place.

Moreover, explicitly typing everything is not necessary for optimal performance.

The Julia compiler is smart and can often infer types perfectly well, without any performance cost.

What we really want to do is

- keep our code simple, elegant and generic
- help the compiler out in situations where it's liable to get tripped up

7.2 Summary and Tips

Use functions to segregate operations into logically distinct blocks.

Data types will be determined at function boundaries.

If types are not supplied then they will be inferred.

If types are stable and can be inferred effectively your functions will run fast.

7.3 Further Reading

A good next stop for further reading is the relevant part of the Julia documentation.