Notes 6: JIT compilation

Chris Paciorek

2025-02-06

Table of contents

Introduction

This document is the sixth of a set of notes, this document focusing on the basics of Just-in-Time (JIT) compilation. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Compilation overview

Compilation and interpreters

Compilation is the process of transforming code from one representation to another. This generally involves transforming high-level, human-readable code to lower level representations that are closer to the instructions the CPU actually performs. For example, the C compiler transforms C code to (binary) machine code (the `.o` or `.so` file(s)). One can then run the binary code directly.

In contrast, R or Python code is executed by an interpreter that parses the code and evaluates it. Generally the interpreter is itself a C program (e.g., CPython). When you are in an interactive environment

(a "REPL"), there is the additional layer of the interactive interface (e.g., RStudio, IPython).

JIT compilation overview

Unlike compilation of standard non-interactive languages like C, C++, and Fortran, the Julia compilation process happens at run-time rather than in advance, hence "just-in-time".

The compilation process involves a series of transformation steps, including:

- parsing
- macro expansion
- type inference
- translation to an intermediate representation (IR)
- translation to LLVM code
- generation of machine code

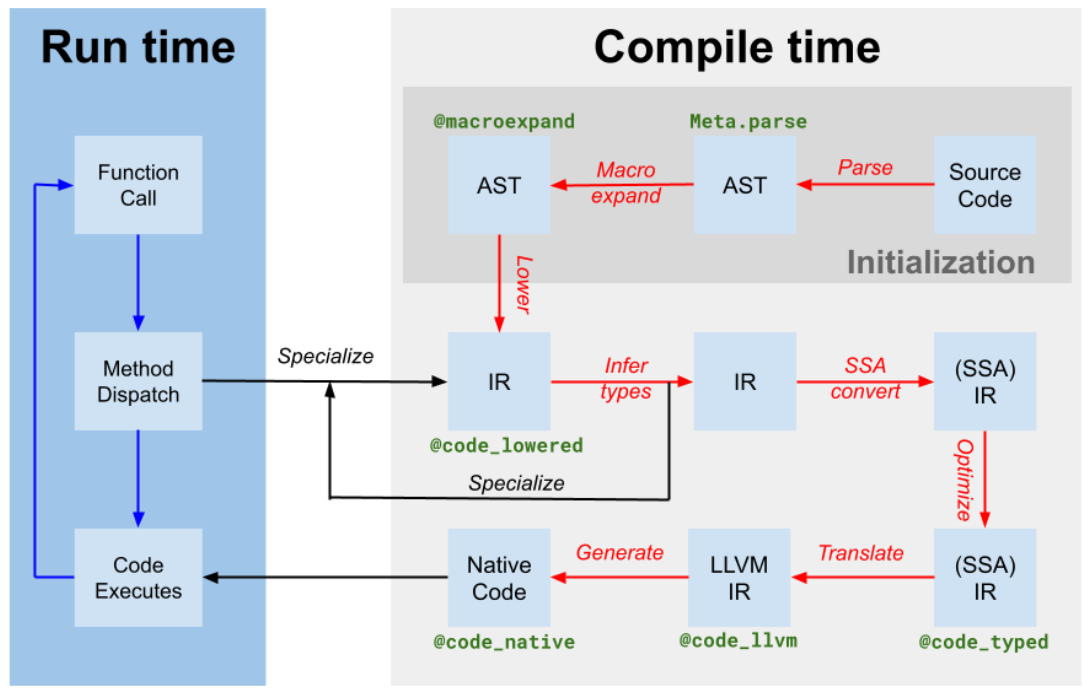

Figure 1: Julia compiler steps (courtesy of the Julia manual)

We'll see at the end of these notes that we can look at the output of the various steps using the macros shown in the diagram.

LLVM

Julia uses LLVM, which provides the middle layers of a compilation system. For example, on a Mac, the default C/C++ compiler is Clang, which is build on LLVM.

LLVM provides an intermediate representation (IR) that is independent of the programming language and can be thought of as a high-level assembly language. By going through LLVM's IR, one can take advantage of LLVM's tools for code optimization.

JIT examples

Multiple dispatch

Julia compiles version of functions for each type of inputs. Let's see that with an example.

```
function mysum(x)
  out=(typeof(x[1]))(0)
  n = length(x)
  for i in 1:n
     out += x[i]
  end
  return out
end

xfloat = [3.1, 5.7, 3.2]
xfloat32 = Float32[3.1, 5.7, 3.2]
xint = [3, 5, 4]

@time mysum(xfloat)
```

```
  0.010197 seconds (2.86 k allocations: 194.250 KiB, 99.75% compilation time)
```

```
12.0
```

```
@time mysum(xfloat)
```

```
  0.000008 seconds (1 allocation: 16 bytes)
```

```
12.0
```

```
@time mysum(xfloat32)
```

```
  0.009549 seconds (2.85 k allocations: 193.719 KiB, 99.74% compilation time)
```

```
11.999999f0
```

```
@time mysum(xint)
```

```
  0.013518 seconds (2.85 k allocations: 193.688 KiB, 99.83% compilation time)
```

```
12
```

For loop speed

Let's explore the timing of Julia looping to better understand the JIT compilation.

We'll use the well-used example of the Monte Carlo approach to estimating π.

Here's a Julia implementation:

```
in_circle = 0;
num_throws = 5000;

# Run Monte Carlo simulation
for _ in 1:num_throws
  # Generate random x and y coordinates between -1 and 1
  local xpos = rand() * 2 - 1.0  # Equivalent to random.uniform(-1.0, 1.0)
  local ypos = rand() * 2 - 1.0

  # Check if point is inside unit circle
  if sqrt(xpos^2 + ypos^2) <= 1.0  # Equivalent to math.hypot()
    in_circle += 1
  end
end

# Estimate PI
pi_estimate = 4 * in_circle / num_throws
```

```
3.124
```

> Exercise
>
> Try the following:
> 1. Time the for loop above without putting it into a function (`@time begin ... end`).
> 2. Put the code within a function and time it using `@time` when running the first time.
> 3. Time it after running it a second time
> 4. Time a vectorized version without the loop (also within a function)
>
> Fill in your timing answers in this Google form and we'll discuss during the next class.

JIT and global versus local variables

Using global variables is much less efficient than using local variables because the type of the global variable could change and that makes it hard to generate simple optimized code.

```
function squaresum(x, y)
  return x^2 + y^2
end
```

```
function squaresum_with_global(x)
  return x^2 + y^2
end
```

```
squaresum_with_global (generic function with 1 method)
```

The intermediate representations

Let's look at the LLVM (intermediate representation) code produced by the JIT compilation process.

The LLVM code for the basic version is simple and in fact we can read and understand it.

```
@code_llvm squaresum(4, 3)
```

```
;  @ In[7]:1 within `squaresum`
define i64 @julia_squaresum_953(i64 signext %0, i64 signext %1) #0 {
top:
;  @ In[7]:2 within `squaresum`
;   @ intfuncs.jl:332 within `literal_pow`
;    @ int.jl:88 within `*`
    %2 = mul i64 %0, %0
    %3 = mul i64 %1, %1
;
;   @ int.jl:87 within `+`
   %4 = add i64 %3, %2
;
  ret i64 %4
}
```

In contrast the LLVM code for the version with the global is quite complicated (and only partially shown below).

```
@code_llvm squaresum_with_global(4)
```

```
;  @ REPL[22]:1 within `squaresum_with_global`
define nonnull {}* @julia_squaresum_with_global_590(i64 signext %0) #0 {
top:
  %1 = alloca [3 x {}*], align 8
  %gcframe2 = alloca [4 x {}*], align 16
  %gcframe2.sub = getelementptr inbounds [4 x {}*], [4 x {}*]* %gcframe2, i64 0, i64 0
  %.sub = getelementptr inbounds [3 x {}*], [3 x {}*]* %1, i64 0, i64 0
  %2 = bitcast [4 x {}*]* %gcframe2 to i8*
  call void @llvm.memset.p0i8.i64(i8* align 16 %2, i8 0, i64 32, i1 true)
  %thread_ptr = call i8* asm "movq %fs:0, $0", "=r"() #6
  %tls_ppgcstack = getelementptr i8, i8* %thread_ptr, i64 -8
  %3 = bitcast i8* %tls_ppgcstack to {}****
[... snip ...]
```

Let's time the two:

```
using BenchmarkTools
y = 3;
@btime squaresum_with_global(4);
@btime squaresum(4, 3);
```

```
  35.269 ns (0 allocations: 0 bytes)
  1.299 ns (0 allocations: 0 bytes)
```

That's a very simple function, so it's very fast on an *absolute* basis even with the global, but clearly very slow *relative* to the non-global version. That makes sense given the additional code in the LLVM code for the global version.

Helping out the compiler

Now let's see that using a `const` or typing the global variable avoids the problem.

```
function squaresum_with_const_global(x)
  return x^2 + z^2
end

const z = 3;
@code_llvm squaresum_with_const_global(4)
```

```
;  @ In[10]:1 within `squaresum_with_const_global`
define i64 @julia_squaresum_with_const_global_1133(i64 signext %0) #0 {
top:
;  @ In[10]:2 within `squaresum_with_const_global`
;   @ intfuncs.jl:332 within `literal_pow`
;    @ int.jl:88 within `*`
    %1 = mul i64 %0, %0
;
;   @ int.jl:87 within `+`
   %2 = add i64 %1, 9
;
  ret i64 %2
}
```

```
@btime squaresum_with_const_global(4);
```

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

sing the constant brings the speed back to the version without a global variable. We can see in the LLVM version of the code that z^2 has been computed and is hard-coded into the code. So it makes sense that it's just as fast as the non-global version (in fact we'd expect it to be faster as there is only one squaring operation).

Here we add a type declaration to the global.

```
w::Int64 = 3
function squaresum_with_typed_global(x)
```

```
  return x^2 + w^2
end

@code_llvm squaresum_with_typed_global(4)
```

```
;  @ In[12]:2 within `squaresum_with_typed_global`
define i64 @julia_squaresum_with_typed_global_1139(i64 signext %0) #0 {
top:
;  @ In[12]:3 within `squaresum_with_typed_global`
;   @ intfuncs.jl:332 within `literal_pow`
;    @ int.jl:88 within `*`
    %1 = mul i64 %0, %0
;
  %w2 = load atomic i64*, i64** inttoptr (i64 140237756770256 to i64**) unordered, align 16
;   @ intfuncs.jl:332 within `literal_pow`
;    @ int.jl:88 within `*`
    %unbox = load i64, i64* %w2, align 8
    %2 = mul i64 %unbox, %unbox
;
;   @ int.jl:87 within `+`
   %3 = add i64 %2, %1
;
  ret i64 %3
}
```

```
@btime squaresum_with_typed_global(4);
```

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

The typed global avoids the performance issue as well. (Mostly –it is a bit slower, presumably because of the retrieval of the global variable.)

Of course from the perspective of code style, avoiding global variables is generally a good idea, even if we're able in this case to avoid a performance problem.

Side note: when I was exploring this, if I change the type of the global variable, it doesn't seem to trigger compilation or change the compiled code. So there is something I'm not understanding.

JIT: intermediate representations

Let's look at the various representations along the Julia compilation process.

```
# using InteractiveUtils
@code_lowered squaresum(3.0, 4.0)
```

```
CodeInfo(
1   %1 = Main.:^
    %2 = Core.apply_type(Base.Val, 2)
    %3 = (%2)()
```

```
    %4 = Base.literal_pow(%1, x, %3)
    %5 = Main.:^
    %6 = Core.apply_type(Base.Val, 2)
    %7 = (%6)()
    %8 = Base.literal_pow(%5, y, %7)
    %9 = %4 + %8
        return %9
)
```

```
@code_typed squaresum(3.0, 4.0)
```

```
CodeInfo(
1   %1 = Base.mul_float(x, x)::Float64
    %2 = Base.mul_float(y, y)::Float64
    %3 = Base.add_float(%1, %2)::Float64
        return %3
) => Float64
```

```
@code_llvm squaresum(3.0, 4.0)
```

```
;  @ In[7]:1 within `squaresum`
define double @julia_squaresum_1533(double %0, double %1) #0 {
top:
;  @ In[7]:2 within `squaresum`
;   @ intfuncs.jl:332 within `literal_pow`
;    @ float.jl:411 within `*`
    %2 = fmul double %0, %0
    %3 = fmul double %1, %1
;
;   @ float.jl:409 within `+`
   %4 = fadd double %2, %3
;
  ret double %4
}
```

Assembly and machine code

Here's the assembly ('native') code, which has a very close relationship with the actual binary machine code.

```
@code_native squaresum(3.0, 4.0)
```

```
    .text
    .file   "squaresum"
    .globl  julia_squaresum_1549            # -- Begin function julia_squaresum_1549
    .p2align    4, 0x90
    .type   julia_squaresum_1549,@function
julia_squaresum_1549:                   # @julia_squaresum_1549
;   @ In[7]:1 within `squaresum`
```

```
# %bb.0:                                # %top
    push    rbp
    mov rbp, rsp
;   @ In[7]:2 within `squaresum`
;    @ intfuncs.jl:332 within `literal_pow`
;     @ float.jl:411 within `*`
    vmulsd  xmm0, xmm0, xmm0
    vmulsd  xmm1, xmm1, xmm1
;
;    @ float.jl:409 within `+`
    vaddsd  xmm0, xmm0, xmm1
;
    pop rbp
    ret
.Lfunc_end0:
    .size   julia_squaresum_1549, .Lfunc_end0-julia_squaresum_1549
;
                                        # -- End function
    .section    ".note.GNU-stack","",@progbits
```

And here's the "binary" version, though it seems like it's still showing a version of the assembly code but with some binary annotation.

```
code_native(squaresum, binary=true)
```

```
; WARNING: This code may not match what actually runs.
    .text
    .file   "squaresum"
    .globl  julia_squaresum_1566            # -- Begin function julia_squaresum_1566
    .p2align    4, 0x90
    .type   julia_squaresum_1566,@function
julia_squaresum_1566:                   # @julia_squaresum_1566
;   @ In[7]:1 within `squaresum`
# %bb.0:                                # %top
    push    rbp                             # encoding: [0x55]
    mov rbp, rsp                        # encoding: [0x48,0x89,0xe5]
    push    r15                             # encoding: [0x41,0x57]
    push    r14                             # encoding: [0x41,0x56]
    push    r13                             # encoding: [0x41,0x55]
    push    r12                             # encoding: [0x41,0x54]
    push    rbx                             # encoding: [0x53]
    and rsp, -32                        # encoding: [0x48,0x83,0xe4,0xe0]
    sub rsp, 96                         # encoding: [0x48,0x83,0xec,0x60]
    mov qword ptr [rsp + 24], rsi       # 8-byte Spill
                                        # encoding: [0x48,0x89,0x74,0x24,0x18]
    vxorps  xmm0, xmm0, xmm0                # encoding: [0xc5,0xf8,0x57,0xc0]
    vmovaps ymmword ptr [rsp + 32], ymm0    # encoding: [0xc5,0xfc,0x29,0x44,0x24,0x20]
```

```
    #APP
    mov rax, qword ptr fs:[0]          # encoding: [0x64,0x48,0x8b,0x04,0x25,0x00,0x00,0x00,0x00]
    #NO_APP
    mov r13, qword ptr [rax - 8]       # encoding: [0x4c,0x8b,0x68,0xf8]
;   @ In[7]:2 within `squaresum`
    mov qword ptr [rsp + 32], 8        # encoding: [0x48,0xc7,0x44,0x24,0x20,0x08,0x00,0x00,0x00]
    mov rax, qword ptr [r13]           # encoding: [0x49,0x8b,0x45,0x00]
    mov qword ptr [rsp + 40], rax      # encoding: [0x48,0x89,0x44,0x24,0x28]
    lea rax, [rsp + 32]                # encoding: [0x48,0x8d,0x44,0x24,0x20]
    mov qword ptr [r13], rax           # encoding: [0x49,0x89,0x45,0x00]
    movabs  rax, 140237579662080           # encoding: [0x48,0xb8,0x00,0xb3,0x1e,0x9b,0x8b,0x7f,0x00
    mov qword ptr [rsp], rax           # encoding: [0x48,0x89,0x04,0x24]
    mov qword ptr [rsp + 8], rdi       # encoding: [0x48,0x89,0x7c,0x24,0x08]
    movabs  rax, 140237579076032           # encoding: [0x48,0xb8,0xc0,0xc1,0x15,0x9b,0x8b,0x7f,0x00
    mov qword ptr [rsp + 16], rax      # encoding: [0x48,0x89,0x44,0x24,0x10]
    movabs  r14, offset ijl_apply_generic  # encoding: [0x49,0xbe,A,A,A,A,A,A,A,A]
                                       #   fixup A - offset: 2, value: ijl_apply_generic, kind: FK_D
    movabs  r15, 140237614986928           # encoding: [0x49,0xbf,0xb0,0xb6,0x39,0x9d,0x8b,0x7f,0x00
    mov r12, rsp                       # encoding: [0x49,0x89,0xe4]
    mov rdi, r15                       # encoding: [0x4c,0x89,0xff]
    mov rsi, r12                       # encoding: [0x4c,0x89,0xe6]
    mov edx, 3                         # encoding: [0xba,0x03,0x00,0x00,0x00]
    vzeroupper                             # encoding: [0xc5,0xf8,0x77]
    call    r14                            # encoding: [0x41,0xff,0xd6]
    mov rbx, rax                       # encoding: [0x48,0x89,0xc3]
    mov qword ptr [rsp + 56], rax      # encoding: [0x48,0x89,0x44,0x24,0x38]
    movabs  rax, 140237579662080           # encoding: [0x48,0xb8,0x00,0xb3,0x1e,0x9b,0x8b,0x7f,0x00
    mov qword ptr [rsp], rax           # encoding: [0x48,0x89,0x04,0x24]
    mov rax, qword ptr [rsp + 24]      # 8-byte Reload
                                       # encoding: [0x48,0x8b,0x44,0x24,0x18]
    mov qword ptr [rsp + 8], rax       # encoding: [0x48,0x89,0x44,0x24,0x08]
    movabs  rax, 140237579076032           # encoding: [0x48,0xb8,0xc0,0xc1,0x15,0x9b,0x8b,0x7f,0x00
    mov qword ptr [rsp + 16], rax      # encoding: [0x48,0x89,0x44,0x24,0x10]
    mov rdi, r15                       # encoding: [0x4c,0x89,0xff]
    mov rsi, r12                       # encoding: [0x4c,0x89,0xe6]
    mov edx, 3                         # encoding: [0xba,0x03,0x00,0x00,0x00]
    call    r14                            # encoding: [0x41,0xff,0xd6]
    mov qword ptr [rsp + 48], rax      # encoding: [0x48,0x89,0x44,0x24,0x30]
    mov qword ptr [rsp], rbx           # encoding: [0x48,0x89,0x1c,0x24]
    mov qword ptr [rsp + 8], rax       # encoding: [0x48,0x89,0x44,0x24,0x08]
    movabs  rdi, 140237580826512           # encoding: [0x48,0xbf,0x90,0x77,0x30,0x9b,0x8b,0x7f,0x00
    mov rsi, r12                       # encoding: [0x4c,0x89,0xe6]
    mov edx, 2                         # encoding: [0xba,0x02,0x00,0x00,0x00]
    call    r14                            # encoding: [0x41,0xff,0xd6]
    mov rcx, qword ptr [rsp + 40]      # encoding: [0x48,0x8b,0x4c,0x24,0x28]
    mov qword ptr [r13], rcx           # encoding: [0x49,0x89,0x4d,0x00]
```

```
    lea rsp, [rbp - 40]                # encoding: [0x48,0x8d,0x65,0xd8]
    pop rbx                            # encoding: [0x5b]
    pop r12                            # encoding: [0x41,0x5c]
    pop r13                            # encoding: [0x41,0x5d]
    pop r14                            # encoding: [0x41,0x5e]
    pop r15                            # encoding: [0x41,0x5f]
    pop rbp                            # encoding: [0x5d]
    ret                                    # encoding: [0xc3]
.Lfunc_end0:
    .size   julia_squaresum_1566, .Lfunc_end0-julia_squaresum_1566
;
                                       # -- End function
    .section    ".note.GNU-stack","",@progbits
```