Julia

Lukas Koschmieder, Kai Neumann

July 22, 2014

Outline

RWTH AACHEN UNIVERSITY

Overview

RWTH AACHEN UNIVERSITY

Language classification

- high-level dynamic language
- designed for technical computing
- emphasis on performance

RWTH AACHEN UNIVERSITY

Motivation

"The most challenging areas of technical programming benefit the least from technical languages."

- dynamic languages
  - convenient and highly productive
  - lack in speed
- compromise (two-tier approach)
  - high-level logic in dynamic language
  - computationally-intensive routines in C or Fortran
- Julia
  - as fast as a statically compiled language
  - as interactive/productive as a dynamic language

RWTH AACHEN UNIVERSITY

Overview

Code representations

1. AST after parsing
2. AST after lowering
3. AST after type inference
4. LLVM IR
5. Assembly code

Layers 1: AST

- abstract syntax tree
- internal representation of Julia code
- specific to Julia
- produced by parsing Julia code
- operators become function calls
- representation is not stored

RWTH AACHEN UNIVERSITY

Layers 1: AST - Example

Infix operator removal

```
julia> 1+2
3
julia> :(1+2)
:(+(1,2))
julia> +(1,2)
3
julia> :(+(1,2))
:(+(1,2))
julia> :(1+2+3+4+5)
:(+(1,2,3,4,5))
julia> :(1+2-3-4+5)
:(+(-(-(+(1,2),3),4),5))
```

Layers 2: Lowered AST

- lowering
  - in general: process of moving from surface syntax (highest) to machine code (lowest)
  - here: transformation of the AST into a simpler version
    - unnesting expressions
    - desugaring syntax
    - simplifing control flow
    - reduction of the instruction set
    - etc.

RWTH AACHEN UNIVERSITY

Layers 2: Lowered AST - Example

for loop

```
julia> function loop(x::Int)
         y = 0
         for i = 1:x
           y += x
         end
         y
       end
```

RWTH AACHEN UNIVERSITY

Layers 2: Lowered AST - Example

for loop lowered to unless/goto

```
1-element Array{Any,1}:
:($(Expr(:lambda, {:x}, {{:y,:#s6,:#s5,:i},{{:x,:Any,0},{:y,:Any
    ,2},{:#s6,:Any,2},{:#s5,:Any,18},{:i,:Any,18}},{}}, quote  #
    none, line 2:
        y = 0 # line 3:
        #s6 = 1
        #s5 = x
        1:
        unless top(<=)(#s6,#s5) goto 2
        i = #s6 # line 4:
        y = +(y,x)
        3:
        #s6 = top(convert)(top(typeof)(#s6),top(+)(1,#s6))
        goto 1
        2:
        0:  # line 6:
        return y
```

Layers 3: Type-inferred AST

- Type inferrence
  - automatic deduction of the type of an expression at compile time
  - also: optimization according to types

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 1

integer definition

```
julia> function foo()
         x = 1
       end
```

string definition

```
julia> function bar()
         x = "bar"
       end
```

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 1

lowered integer definition

```
1-element Array{Any,1}:
 :($(Expr(:lambda, {}, {{:x},{{:x,Any,18}},{}},
quote   # none, line 2:
        x = 1
        return 1
    end)))
```

lowered string definition

```
1-element Array{Any,1}:
 :($(Expr(:lambda, {}, {{:x},{{:x,Any,18}},{}},
quote   # none, line 2:
        x = "bar"
        return "bar"
    end)))
```

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 1

type-inferred integer definition

```
1-element Array{Any,1}:
 :($(Expr(:lambda, {}, {{:x},{{:x,Int32,18}},{}},
quote  # none, line 2:
        x = 1
        return 1
    End)))
```

type-inferred string definition

```
1-element Array{Any,1}:
 :($(Expr(:lambda, {}, {{:x},{{:x,ASCIIString,18}},{}},
quote  # none, line 2:
        x = "bar"
        return "bar"
    end)))
```

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 2

star operator

```
julia> function foobar(x)
         x = x * x
       end
```

star operator on integers

```
julia> foobar(256)
65536
```

star operator on strings

```
julia> foobar("256")
"256256"
```

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 2

lowered star operator

```
1-element Array{Any,1}:
 :($(Expr(:lambda, {:x}, {{:#s5},{{:x,:Any,18},{:#s5,:Any
     ,18}},{}}, quote  # none, line 2:
         #s5 = *(x,x)
         x = #s5
         return #s5
     end)))
```

RWTH AACHEN UNIVERSITY

Layers 3: Type-inferred AST - Ex. 2

type-inferred star operator on integers

```
(...)
    #s5 = top(box)(Int,top(mul_int)(x::Int32,x::Int32))::Int32
    x = #s5::Int32
    return #s5::Int32
(...)
```

type-inferred star operator on strings

```
(...)
    #s5 = string(x::String,x::String)::Union(UTF8String,
        ASCIIString)
    x = #s5::Union(UTF8String,ASCIIString)
    return #s5::Union(UTF8String,ASCIIString)
(...)
```

Layers 4: LLVM IR

- LLVM IR
  - LLVM intermediate representation
  - not specific to Julia
  - passed to LLVM compiler
  - requires types

RWTH AACHEN UNIVERSITY

Layers 4: LLVM IR - Example

star operator on integers

```
define i32 @julia_foobar926(i32) {
top:
%1 = mul i32 %0, %0, !dbg !4746
ret i32 %1, !dbg !4746
}
```

star operator on strings

```
define %jl_value_t* @julia_foobar927(%jl_value_t*, %jl_value_t**,
      i32) {
top:
%3 = alloca [6 x %jl_value_t*], align 4
%.sub = getelementptr inbounds [6 x %jl_value_t*]* %3, i32 0, i32
(51 more lines)
}
```

RWTH AACHEN UNIVERSITY

Layers 5: Assembler code

- Assembler code
  - optimized machine-specific code
  - provided by LLVM compiler

star operator on integers

```
.text
Filename: none
Source line: 2
push EBP
mov  EBP, ESP
mov  EAX, 1
pop  EBP
ret
```

RWTH AACHEN UNIVERSITY

Overview

Multiple dispatch

"Multiple dispatch is the ability to choose which version of a function to call based on the run-time type of the arguments passed to the function call."

- object-oriented programming
  - single dispatch
  - dispatch based only on 1st argument (object type)
- Java/C++ (single dispatch)
  - object.function(arg1, arg2, ...)
- Julia (multiple dispatch)
  - function(object, arg1, arg2, ...)

RWTH AACHEN UNIVERSITY

Multiple dispatch

Asteroids collision system in Julia

```
function collide_with(x::Asteroid,  y::Asteroid ) { ... }
function collide_with(x::Asteroid,  y::Spaceship) { ... }
function collide_with(x::Spaceship, y::Asteroid ) { ... }
function collide_with(x::Spaceship, y::Spaceship) { ... }
```

- useful for mathematic code from the programer's point of view

a+b in Julia

```
function +(a::Int, b::Int) = {
   ...
}
+(a,b)
```

RWTH AACHEN UNIVERSITY

The key ingrediences of performance

1. Rich type information provided naturally by multiple dispatch

The key ingrediences of performance

1. Rich type information provided naturally by multiple dispatch
2. Aggressive code specialization against run-time types

The key ingrediences of performance

1. Rich type information provided naturally by multiple dispatch
2. Aggressive code specialization against run-time types
3. JIT compilation using the LLVM compiler framework

RWTH AACHEN UNIVERSITY

Overview

RWTH AACHEN UNIVERSITY

Julia Multiple Dispatch

- Julia choses which function is called depending on its parameters' types
- the @which macro can be used to see which function julia will call on which parameters

Integer Multiplication

```
julia> @which 1*2
*(x::Int64,y::Int64) at int.jl:55
```

Vector Multiplication

```
julia> A=ones(1,10); @which A*2
*(A::AbstractArray{T,N},B::Number) at abstractarray.jl:340
```

RWTH AACHEN UNIVERSITY

Julia Multiple Dispatch: IR

Scalar Multiplication: $Int64 \times Int64$

```
define i64 @"julia_*1396"(i64, i64) {
top:
  %2 = mul i64 %1, %0, !dbg !6861
  ret i64 %2, !dbg !6861
}
```

Vector Multiplication: $Vector \times Int64$

```
define %jl_value_t* @"julia_*1378"(%jl_value_t*, i64) {
top:
  %2 = call %jl_value_t* @"julia_.*1380"(%jl_value_t* %0, i64 %1)
       , !dbg !6861
  ret %jl_value_t* %2, !dbg !6861
}
```

RWTH AACHEN UNIVERSITY

LLVM IR: Julia vs clang++

Increment: Julia

```
define i64 @julia_increment(i64) {
top:
  %1 = add i64 %0, 1, !dbg !6898
  ret i64 %1, !dbg !6898
}
```

Increment: Clang++

```
define i32 @_Z9incrementi(i32 %a) #0 {
entry:
  %a.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %0 = load i32* %a.addr, align 4
  %add = add nsw i32 %0, 1
  ret i32 %add
}
```

RWTH AACHEN UNIVERSITY

Overview

Performance: Microbenchmarks

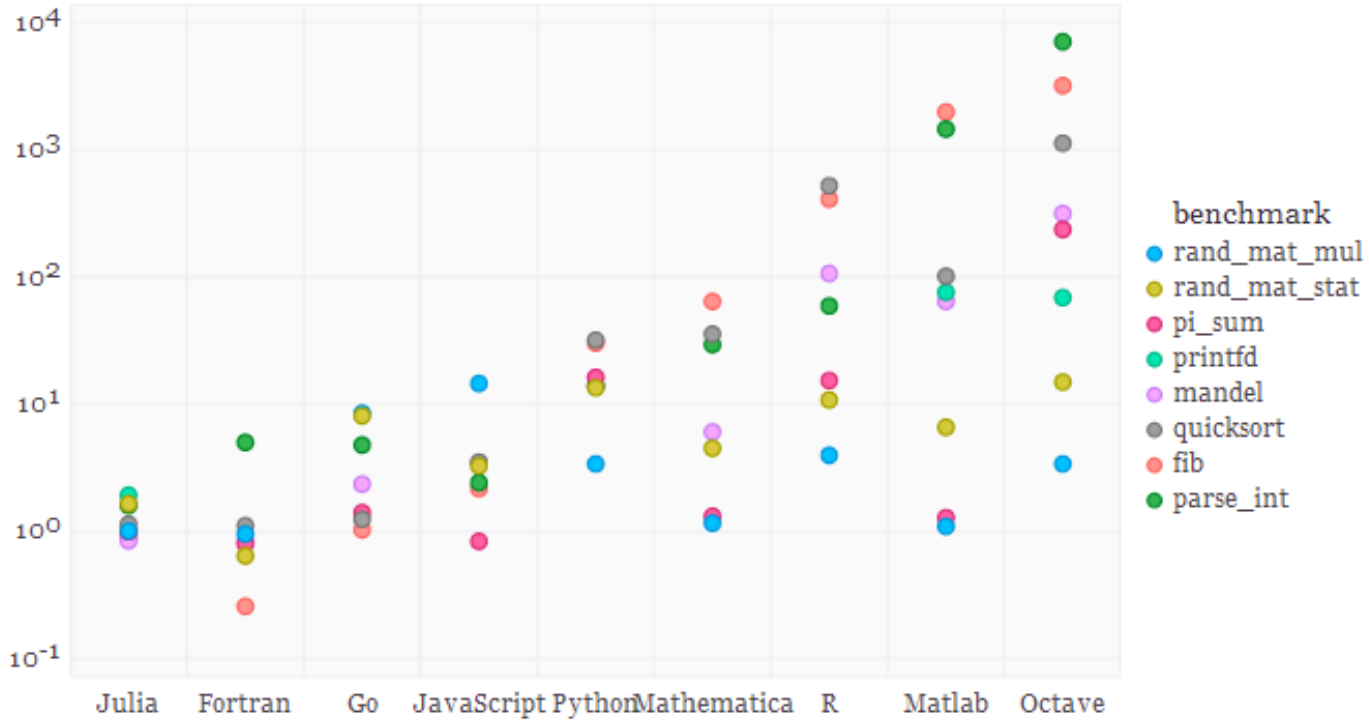

Figure : Benchmark times relative to C, http://julialang.org, https://github.com/JuliaLang/julia/tree/master/test/perf/micro

RWTH AACHEN UNIVERSITY

Performance (2)

- Julia performance within factor of 2 compared to native C
- Julia Startup-Time not included
- Julia caches native code, first call to function is slower, but only min is measured
- Also: Other languages often faster in special cases

RWTH AACHEN UNIVERSITY

Performance (3)

CumProd in MATLAB 2014a

```
A=ones(100,1000000);b=0.9;tic;cumprod(A*b);toc
Elapsed time is 0.679127 seconds.
```

CumProd in Julia

```
A=ones(100,1000000);b=0.9;@elapsed cumprod(A*b)
2.988972363
```

$\Rightarrow$ Speedup = 4,4

RWTH AACHEN UNIVERSITY

Overview

RWTH AACHEN UNIVERSITY

Parallelism

- Message-Passing Model
- Unlike MPI, Communication is 'one-sided'
- $\Rightarrow$ View only from Master thread, call external functions and fetch results
- Macros for distributed loops

Parallel for loop

```
nheads = @parallel (+) for i=1:100000000
  int(randbool())
end
```

RWTH AACHEN UNIVERSITY

Parallelism(2)

Spawning functions on other processes

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.1082421641130486 1.1379823387792316
1.1237629270635507 1.1875049791660717
```

RWTH AACHEN UNIVERSITY

Overview

RWTH AACHEN UNIVERSITY

Language Features

- Package Manager (Packages for Plotting, Statistics, Websockets, etc)
- Integration (C, Fortran Calls)
- Free & Open Source
- Active Community
- High-Level syntax

RWTH AACHEN UNIVERSITY

Overview

RWTH AACHEN UNIVERSITY

Conclusion

- Julia is a dynamic high-level language using LLVM as JIT compiler
- 5 stages/layers to transform julia code into assembly code
- Julia aims on the gap between dynamic and statically compiled languages
- Even with nice performance, still a tradeoff between productivity and performance
- Does not outperform every other language in every category

RWTH AACHEN UNIVERSITY

More Information

- Interactive web-based Julia (including Tutorial): http://forio.com/julia/repl/
- YouTube: JuliaLanguage
- http://julialang.org/

RWTH AACHEN UNIVERSITY

References I


Jeff Bezanson and Stefan Karpinski and Viral B. Shah and Alan Edelman
Julia: A Fast Dynamic Language for Technical Computing
CoRR, abs/1209.5145, 2012.


http://julialang.org/


https://github.com/JuliaLang/julia


http://media.readthedocs.org/pdf/julia/latest/julia.pdf


http://blog.leahhanson.us/julia-introspects.html


http://scientopia.org/blogs/goodmath/2014/02/04/everyone-stop-implementing-programming-languages-right-now-its-been-solved/