Julia- Yet Another Lisp

Albuquerque Lisp Club - 2014-08-28

Speaker Not An Expert

Why I am Got Interested

- Motivation: Scientific Computation
- A couple of blogposts by Gradydon: “Goldilocks Languages”
  - Ousterhout’s Dichotomy: high level/low level languages (TCL/C in his case)
  - Yuck
- Common Lisp/Scheme span both, but some things “missing” or “not common”:
  - Support for graphics, data formats (HDF5), etc.
  - Leverage: easy integration with C/Fortran/Python etc.
  - Support for parallel computing (we closet >100$/hr, CPU 0.05$/hr)
  - Networking libraries
  - Notebook interface

Is Julia a Lisp? Lispy Features

- It has a REPL
- It has GC
- Simple, uniform syntax
- It’s homoiconic
- Everything is an expression
- Incremental compilation using LLVM (more later!)
- It has macros (will demonstrate)
- Written at MIT, open source & development (github)

Homoiconic: Wikipedia

In computer programming, **homoiconicity** (from the Greek words *homo* meaning *the same* and *icon* meaning *representation*) is a property of some programming languages in which the program structure is similar to its syntax, and therefore the program's internal representation can be inferred by reading the text's layout.[1] If a language is homoiconic, it means that the language text has the same structure as its abstract syntax tree (i.e. the AST and the syntax are isomorphic). This allows all code in the language to be accessed and transformed as data, using the same representation.

What the Creators Say

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

- Since Feb 2012- currently v0.3, but already useful.
- 11000 C, 4000 C++, 3500 Scheme (as of sept 2012, no libs)

What's It Look Like?

```
function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i = 1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P.'*P)^4)
        w[i] = trace((Q.'*Q)^4)
    end
    std(v)/mean(v), std(w)/mean(w)
end
```

```
function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
```

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

More Examples: Types

```
julia> type Foo
         bar
         baz::Int
         qux::Float64
       end
```

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.",23,1.5)

julia> typeof(foo)
Foo (constructor with 2 methods)
```

```
julia> names(foo)
3-element Array{Symbol,1}:
 :bar
 :baz
 :qux
```

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

```
julia> foo.qux = 2
2.0
```

```
immutable Complex
  real::Float64
  imag::Float64
end
```

```
julia> typeof((1,"foo",2.5))
(Int64,ASCIIString,Float64)
```

Using LLVM

- LLVM originally meant “Low Level Virtual Machine”
- Now compiler infrastructure, ***language agnostic,*** through run time (jit - just in time compilation)
  - still optimizing on VM- but much more
- (Winner 2012 ACM Software System Award)
- Replacing gcc in many places- e.g. Apple

Consequences of LLVM (1/3)

- Cross Language:
  - Call C, etc. directly

```
julia> t = ccall( (:clock, "libc"), Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

```
function gethostname()
  hostname = Array(Uint8, 128)
  ccall( (:gethostname, "libc"), Int32,
         (Ptr{Uint8}, Uint),
         hostname, length(hostname))
  return bytestring(convert(Ptr{Uint8}, hostname))
end
```

```
function compute_dot(DX::Vector{Float64},
DY::Vector{Float64})
  assert(length(DX) == length(DY))
  n = length(DX)
  incx = incy = 1
  product = ccall( (:ddot_, "libLAPACK"),
                   Float64,
                   (Ptr{Int32}, Ptr{Float64},
Ptr{Int32}, Ptr{Float64}, Ptr{Int32}),
                   &n, DX, &incx, DY, &incy)
  return product
end
```

Consequences of LLVM (2/3)

- Familiar memory model: symbols ref. typed objects
- Types in Julia
  - boxing/unboxing costs?
    - "User-defined types are as fast and compact as built-ins"
- Almost all functions are generic functions
  - Multi-methods/Multiple Dispatch "at run time"

Consequences of LLVM (3/3)

- JIT support:
  - Optimize on the fly, incremental compilation
  - type inferences allows everything to be a multi-method/multiple dispatch
  - Very good performance: Comparable to C/Fortran
  - *Without requiring everything to be an array*

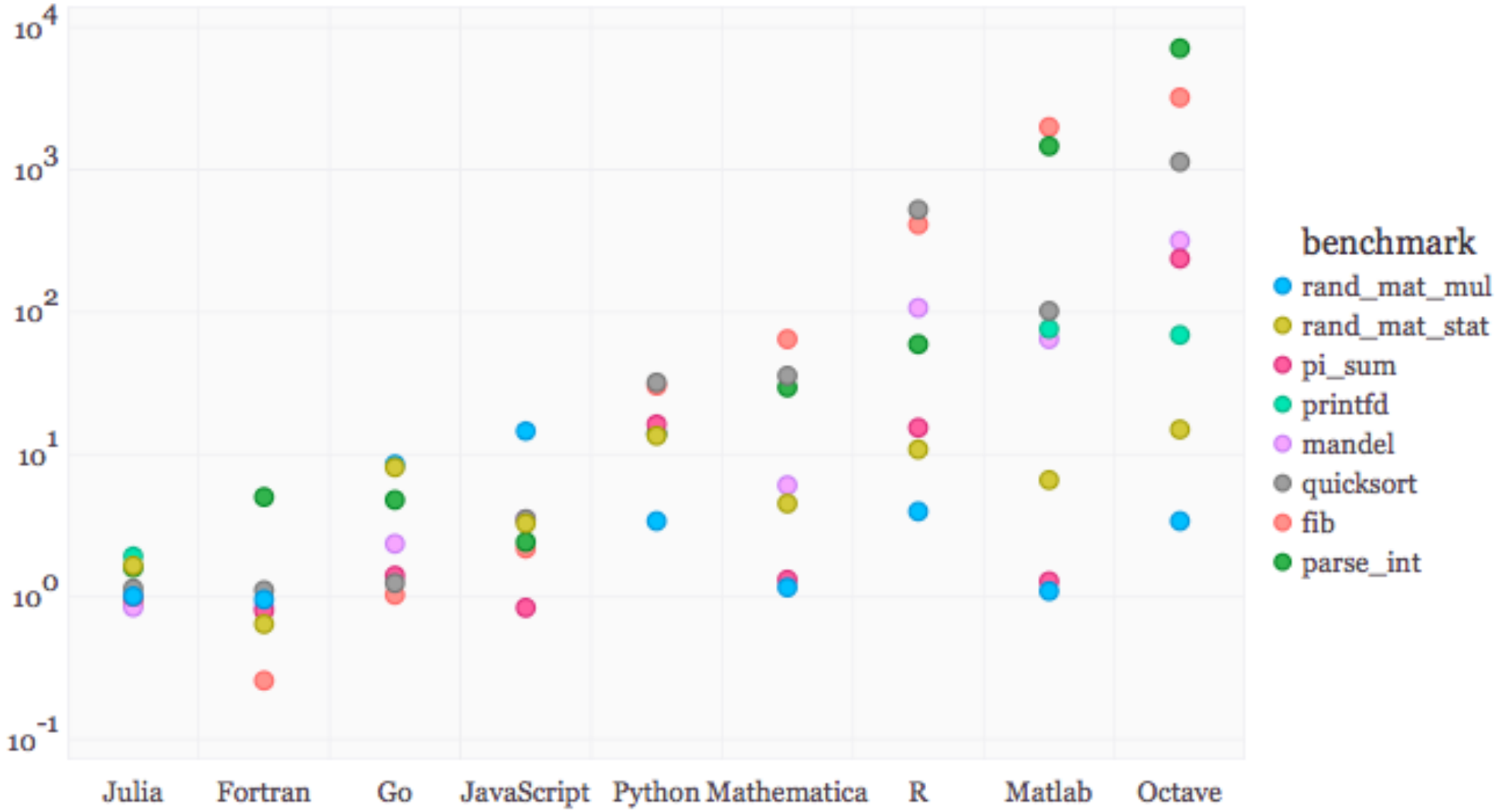

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

C compiled by gcc 4.8.1, taking best timing from all optimization levels (-O0 through -O3). C, Fortran and Julia use OpenBLAS v0.2.8. The Python implementations of `rand_mat_stat` and `rand_mat_mul` use NumPy (v1.6.1) functions; the rest are pure Python implementations. Plot created with Gadfly and IJulia from this notebook.

Macros

Using:

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

Defining:

```
macro name(expr1, expr2, ...)
    ...
    return resulting_expr
end
```

Example:
(showing Hygiene)

```
macro time(ex)
  return quote
    local t0 = time()
    local val = $ex
    local t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    val
  end
end
```

Some Design Features (Costanza)

- Lisp-1, like Scheme, not a Lisp-2
- All variables are lexically scoped- no dynamic variables
- All mathematical functions are generic, can be extended
- Strings are unicode
- return multiple values
- Module system, with fairly well integrated installation

References

- "Learn X in Y Minutes" where X = Julia
  - http://learnxinyminutes.com/docs/julia/
- Language Website
  - http://julialang.org
- Original Paper
  - http://arxiv.org/abs/1209.5145
- Pascal Costanza's Highly Opinionated Blog
  - http://p-cos.blogspot.com/2014/07/a-lispers-first-impression-of-julia.html
- Graydon: Goldilocks Languages
  - http://julialang.org/