A Fast Dynamic Language for Technical Computing

Stefan Karpinski, Jeff Bezanson, Viral Shah, Alan Edelman
What is a “technical/numerical” language?

An obvious answer:

- specialized for numerical work

Matlab:
  - everything is a complex matrix

R (and S before it):
  - allow “NA” values everywhere
  - data frame as basic data type

Mathematica:
  - symbolic rewriting everywhere

NumPy:
  - typed arrays for Python
Are C and Scheme numerical?

Scheme R6RS spec:
- 20% numerical

C99 spec:
- 20% numerical
That's funny...

**Numerical languages** are strangely diverse

**General languages** are strangely numerical

What’s going on here?
The “niche hypothesis”

Numerical computing is still an **under-generalized** niche

- each language picks a **different way** of specializing numerically
  
  (also happens to be the oldest programming language niche – Fortran)

**Hypothesis:**

- many **diverse languages** in this niche can be replaced
- by a **single** sufficiently powerful, general-purpose language
History

**Text processing** was a niche with a similar variety of languages

- SNOBOL, SPITBOL, COMIT, TRAC, TTM, Icon, Unicon, sed, awk, Perl4

  lot’s of **different views** of text processing and how to specialize for it

You don’t see much of these anymore

- people use **one** of Python, Ruby, or Perl5 instead

  (we still use sed and awk sometimes, but **could** use Perl/Python/Ruby)

A few general languages that **support** text processing

- replaced diverse languages that **specialized** in some aspect of it
Text processing was **diverse & hard.**

Now it’s **unified & easy.**

Can we do this for **numerical computing?**
We believe the answer is “yes”

Julia is our attempt to do this.
Before we go further

Let’s actually see some code.
function randmatstat(t,n)
    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 qsort!(a,lo,hi)
    i, j = lo, hi
    while i < hi
        pivot = a[(lo+hi)>>>1]
        while i <= j
            while a[i] < pivot; i = i+1; end
            while a[j] > pivot; j = j-1; end
            if i <= j
                a[i], a[j] = a[j], a[i]
                i, j = i+1, j-1
            end
        end
        if lo < j; qsort!(a,lo,j); end
    end
    lo, j = i, hi
end
return a
end
immutable ModInt\{n\} <: Integer
    k::Int
    ModInt(k) = new(mod(k,n))
end

-{n}(a::ModInt\{n\}) = ModInt\{n\}(-a.k)
+{n}(a::ModInt\{n\}, b::ModInt\{n\}) = ModInt\{n\}(a.k+b.k)
-{n}(a::ModInt\{n\}, b::ModInt\{n\}) = ModInt\{n\}(a.k-b.k)
*{n}(a::ModInt\{n\}, b::ModInt\{n\}) = ModInt\{n\}(a.k*b.k)

convert\{n\}(::Type\{ModInt\{n\}\}, i::Int) = ModInt\{n\}(i)
promote_rule\{n\}(::Type\{ModInt\{n\}\}, ::Type\{Int\}) = ModInt\{n\}

show\{n\}(io::IO, k::ModInt\{n\}) = print(io, "$(k.k) mod $n")
showcompact(io::IO, k::ModInt) = print(io, k.k)
Why are numbers hard?

**Syntax**

- numerical operators tend use **infix syntax**

**Semantics**

- numerical operators are usually not “just functions”
- things like “+” and indexing are **highly polymorphic**
- **special promotion** of arguments to a common type
- need **compact arrays** (of numbers at least)
- numbers are naturally **immutable**
Other things that scientists want...

- extreme **convenience** – things Just Work™
- code that looks like **pseudocode**
- massive **standard library**
- top **performance**
Julia design overview

high-level & **dynamic**

expressive **type-system**

- parametric, dependent, invariant
- concrete types are final
  - but large abstract super-type hierarchy
  - generic programming with abstract types
- **unobtrusive** – don’t need to mention types

metaprogramming

- **homoiconic**: code represented as data
  - can be constructed, manipulated, eval’ed
- **macros**: `@time sleep(1)`

ubiquitous **multiple dispatch**

- everything is a **generic function**
  - even basic performance-critical functions
- quantified methods (think templates)
- diagonal dispatch

concurrency & parallelism

- lightweight **coroutine**-based I/O
- distributed global address space
  - first-class remote references
  - easy to run code on a cluster of instances
- we’re working on **multithreading**
Dynamic typing

Dynamic typing is hugely popular for numerical environments:

- exploratory, interactive, **tangible**
- “**customer is always right**”

Julia has a type system, but no **static type checking**

- Leah Hanson observed while learning Julia:
  
  “**I like that Julia uses the type system in all the ways that don't end with the programmer arguing with the compiler.**”

- not checking types can allow a more sophisticated type system
Two language compromise

People love dynamic environments

- for data analysis and exploration
- but dynamism and performance are at odds

A standard compromise:

- slow code in convenient dynamic language (Matlab, Python, R)
- fast code in static, low-level language (C, C++, Fortran)

Creates a huge impediment to development

- continually breaking the abstraction barrier = poor abstraction
Graydon Hoare (creator of Rust) wrote [http://goo.gl/zQRGu6]:

“Julia, like Dylan and Lisp before it, is a Goldilocks language. It is trying to span the entire spectrum of its target users’ needs, from numerical inner loops to glue-language scripting to dynamic code generation and reflection.”

Goldilocks languages are the opposite of two-language systems:

- do everything in **one language** – both low level and high level work
- define the language **in itself** and give users just as much **power**
“[R]eports ... indicate that Julia gives rather significant boosts over Matlab/R, sometimes by even more than the benchmarks might suggest. That was surprising to me, since I expected the gap to be largest for benchmarks.

…

[O]ne common factor was fairly sizable (but not ridiculous) memory requirements; perhaps Julia's ability to manage memory in a more fine-grained fashion pays major dividends for such problems.”

– Tim Holy, WUSTL
http://goo.gl/r6qwz
Simplex Benchmarks

https://github.com/mlubin/SimplexBenchmarks

- Benchmark of some important operations:

<table>
<thead>
<tr>
<th></th>
<th>Julia</th>
<th>C++</th>
<th>C++bnd</th>
<th>Matlab</th>
<th>PyPy</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp. mat-sp. vec</td>
<td>1.29</td>
<td>0.90</td>
<td>1.00</td>
<td>5.79</td>
<td>19.20</td>
<td>417.16</td>
</tr>
<tr>
<td>Sp. vector scan</td>
<td>1.59</td>
<td>0.96</td>
<td>1.00</td>
<td>13.98</td>
<td>13.81</td>
<td>48.39</td>
</tr>
<tr>
<td>Sp. axpy</td>
<td>1.85</td>
<td>0.70</td>
<td>1.00</td>
<td>19.12</td>
<td>9.21</td>
<td>78.65</td>
</tr>
</tbody>
</table>

- C++bnd = C++ with bounds checking
- Execution times relative to C++bnd
Finite element programming

Comparison by Amuthan Ramabathiran [http://goo.gl/SRciE]:

- **FEniCS**
  
  “collection of software for high level finite element code development written in Python and C++”

- **FreeFem++**
  
  “partial differential equations solver written in C++ with its own DSL (Domain Specific Language) with a C++ like syntax.”

- **Julia FEM**, simple solver
  
  “Thanks to Julia’s elegant syntax the code is largely self-explanatory.”
Finite element programming
“[W]hat is really interesting about Julia is the relative ease with which various strategies can be implemented and tested without leading to code swell, while at the same time resulting in high performance code.

...  

Julia appears to be a very good choice for developing research oriented finite element software that is both fast and easy to develop.”

– Amuthan Ramabathirian
http://goo.gl/SRciE
How does Julia go fast?

There are many fast dynamic language implementations these days

- JavaScript V8, LuaJIT, PyPy, etc.

Julia doesn’t work the way these do at all

- “Julia does static compilation at run time” – Carl Bolz, PyPy core developer

Basically, we’ve cheated

- made key design choices that make it much easier to make things fast

  native data types (machine ints, floats, etc.)
  type annotations; type stability in standard libraries
  immutable types; all concrete types are final
  multiple dispatch
Collatz

function collatz(n)
    k = 0
    while n > 1
        n = isodd(n) ? 3n+1 : n>>1
        k += 1
    end
    return k
end

The Collatz conjecture:

- for all \( n \geq 0 \) the function \( \text{collatz}(n) \) terminates
function collatz_up_to(m)
    c = fill(-1,m)
    c[1] = 0
    for n = 2:m
        n', d = n, 0
        while n' > length(c) || c[n'] < 0
            n' = isodd(n') ? 3n'+1 : n'>>1
            d += 1
        end
        d += c[n']
        while n > length(c) || c[n] < 0
            n <= length(c) && (c[n] = d)
            n = isodd(n) ? 3n+1 : n>>1
            d -= 1
        end
    end
    return c
end
Other key performance tricks

Run-time (just-in-time) code generation using LLVM

› aggressive specialization on runtime types

Very clever data-flow type inference (not Hindley-Milner)

› http://localhost:8998/notebooks/dataflow_type_inference.ipynb

Jeff Bezanson is a true performance artist :-(
But Julia isn’t really about performance

The benchmarks are what grab people, but...

- the real killer is writing **high-level generic code** that runs fast
- and **composing unrelated code** smoothly (and efficiently)

Sounds esoteric, but **multiple dispatch** is crucial

- choose implementation based on on **all arguments**, not just the first
  - trivial to plug in code for efficient **special cases**
  - easy to apply **existing code** to **new types**
  - easy to apply **new code** to **existing types**
Multiple dispatch

What is multiple dispatch?

- **dispatch**: choose method based on runtime types, not static types
- **multiple**: based on all arguments, not just the receiver

Written as **function application**:

- \( f(a,b,c) \) ← like this
- \( a.f(b,c) \) ← not this

Multiple dispatch ≠ method overloading
Multiple dispatch in action

Let’s go to the IJulia Notebook:

Multiple dispatch in Ruby

Arithmetic operators:

<table>
<thead>
<tr>
<th>Number + Number</th>
<th>String + String</th>
<th>Array + Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number - Number</td>
<td>Time - Time</td>
<td>Time - Number</td>
</tr>
<tr>
<td>Number * Number</td>
<td>Array * Integer</td>
<td>Array * String</td>
</tr>
<tr>
<td>Integer &lt;&lt; Integer</td>
<td>String &lt;&lt; String</td>
<td>String &lt;&lt; Integer</td>
</tr>
</tbody>
</table>

Arrays, Hashes & Strings:

<table>
<thead>
<tr>
<th>(Array</th>
<th>Hash).fetch(index,default</th>
<th>block)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Array</td>
<td>Hash).new(object</td>
<td>block)</td>
</tr>
<tr>
<td>(Array</td>
<td>Hash)[int</td>
<td>range]</td>
</tr>
<tr>
<td>(Array</td>
<td>Hash)[int</td>
<td>range]=</td>
</tr>
<tr>
<td>Array.slice(int</td>
<td>range)</td>
<td>String.slice(int</td>
</tr>
<tr>
<td>Array.slice!(int</td>
<td>range)</td>
<td>String.slice!(int</td>
</tr>
</tbody>
</table>

Just Strings:

<table>
<thead>
<tr>
<th>String.index(string</th>
<th>int</th>
<th>regex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>String.rindex(string</td>
<td>int</td>
<td>regex)</td>
</tr>
<tr>
<td>String.sub(pattern,replacement</td>
<td>block)</td>
<td></td>
</tr>
<tr>
<td>String.sub!(pattern,replacement</td>
<td>block)</td>
<td></td>
</tr>
<tr>
<td>String.gsub(pattern,replacement</td>
<td>block)</td>
<td></td>
</tr>
<tr>
<td>String.gsub!(pattern,replacement</td>
<td>block)</td>
<td></td>
</tr>
</tbody>
</table>
Multiple dispatch in English

Related meanings:

- “she goes (home|away)”  
  \[ \text{go} \left( \text{subj::Noun}, \text{where::PlaceAdverb} \right) \]
- “it went (wrong|well)”  
  \[ \text{go} \left( \text{subj::Noun}, \text{how::MannerAdverb} \right) \]

Default arguments:

- “go (home|away|well)”  
  \[ \text{go} \left( \text{adv::Adverb} \right) = \text{go} \left( \text{Person("addressee")}, \text{adv} \right) \]
- “he goes”  
  \[ \text{go} \left( \text{subj::Noun} \right) = \text{go} \left( \text{subj, PlaceAdverb("somewhere")} \right) \]
- “go”  
  \[ \text{go}() = \text{go} \left( \text{PlaceAdverb("somewhere")} \right) \]
Open source & friendly

Julia and most of its packages are

- MIT-licensed
- hosted on GitHub
  - built-in package manager
  - integrated with CI and coverage services (Travis & Coveralls.io)

Active, friendly and helpful community

- helpful for new and veteran programmers, alike
- huge expertise in an breadth of technical subjects
More than just a new language

Julia is a place for programmers, physical scientists, social scientists, computational scientists, mathematicians, and others to pool their collective knowledge in the form of code.