Julia language

v0.6 released this week
• script (matlab-like) syntax and simplicity

• nearly C-like performance without vectorization

• A very deep language. Deep type system, metaprogramming.

• easy calling of C, Python
julia> function mysum(a)
    sum = 0
    for i=1:length(a)
        sum = sum + a[i]
    end
    return sum
end
mysum (generic function with 1 method)

julia> sum([2 3 4])
9
It is comparing Finite Element solver, which is an often used algorithm in material research and therefore represents a relevant use case for Julia.

<table>
<thead>
<tr>
<th>N</th>
<th>JULIA</th>
<th>FENICS(PYTHON + C++)</th>
<th>FREEFEM++(C++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>121</td>
<td>0.99</td>
<td>0.67</td>
<td>0.01</td>
</tr>
<tr>
<td>2601</td>
<td>1.07</td>
<td>0.76</td>
<td>0.05</td>
</tr>
<tr>
<td>10201</td>
<td>1.37</td>
<td>1.00</td>
<td>0.23</td>
</tr>
<tr>
<td>40401</td>
<td>2.63</td>
<td>2.09</td>
<td>1.05</td>
</tr>
<tr>
<td>123201</td>
<td>6.29</td>
<td>5.88</td>
<td>4.03</td>
</tr>
<tr>
<td>251001</td>
<td>12.28</td>
<td>12.16</td>
<td>9.09</td>
</tr>
</tbody>
</table>

(taken from codeproject.)

These are remarkable results, considering that the author states it was not a big effort to achieve this. After all, the other libraries are established FEM solvers written in C++, which should not be easy to compete with.
I did an experiment with calculating a numerical heat transfer (or diffusion) problem in 2D with R, Julia and C++ codes. The problem is like the one in this blog post I have written https://physicscomputingblog.wordpress.com/solution-of-pdes-part-3-2d-diffusion-problem/.

I made a square computational domain, which contained N x N discrete cells, where N was given values 30, 37, 45, 52 and 60 on different runs. The method that was used was implicit finite differencing. The number of timesteps was only 10 in all runs.

The C++ code used simple Gauss-Jordan elimination taken from the book "Numerical Recipes in C", and the Ubuntu C++ compiler was run with parameters "g++ -ffast-math -O3". There was no attempt made to use parallel processing, or to account for the sparseness of the linear system. The matrix inverse was computed only on the first time step, and simple matrix-vector multiplication was used in consecutive time steps.

The R code used the in-built "solve(A,b)" function for solving the system of equations.

The Julia code uses the backslash operator "A\b" for solving the system.

The computation times used by the three codes (not including compilation time) are plotted below for the runs done on my own (slow) laptop (AMD E2-3800 APU with Radeon(TM) HD Graphics x 4).
The problem with single-program benchmarks
my_square(x) = x^2

Thus we don't need to restrict the types we allow in functions in order to get performance. That means that

```
my_restricted_square(x::Int) = x^2
```

is no more efficient than the version above, and actually generates the same exact compiled code:

```
@code_llvm my_restricted_square(1)

define i64 @julia_my_restricted_square_72686(i64) #0 {
  top:
    %1 = mul i64 %0, %0
  ret i64 %1
}
```
Type-Dispatch Design: Post Object-Oriented Programming for Julia

May 29 2017 in Julia, Programming | Tags: | Author: Christopher Rackauckas

In this post I am going to try to explain in detail the type-dispatch design which is used in Julia.

``` julia
my_square(x) = x^2
```

then we see that this function will be efficient for the types that we give it. Looking at the generated code:

``` julia
@code_llvm my_square(1)
define i64 @julia_my_square_72669(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```
@code_llvm my_square(1)
define i64 @julia_my_square_72669(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}

@code_llvm my_square(1.0)
define double @julia_my_square_72684(double) #0 {
top:
  %1 = fmul double %0, %0
  ret double %1
}
TYPE STABILITY

- @code_warntype
- repl code is **not** compiled

```plaintext
function bad(i)
    i = i + 1
    i = “bad”
end
```


#include <iostream>
#include <cstdio>
#include <ctime>
#include <stdio.h>

main()
{
    std::clock_t start;
    double duration;
    double x = 1000.0;
    start = std::clock();
    for(int n=0; n<10; n++)
    {
        x*=0.9999999;
    }
    duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
    std::cout<<"x="<< x <<"\n";
    std::cout<<"time (s): "<< duration <<"\n";
}

result: x=0.0453999

time (s): 0.341655

Then an equivalent Julia code:

```julia
x = 1000.0
for k in 1:10000000
    x*=0.9999999
end

result: x=0.04539990730150107

time (s): 8.537868758
```

So quite a large difference in favor of C++ with this kind of calculation, at least. I'm not sure if telling the Julia to use less significant figures would make it faster.
The Julia example is missing something important: Julia only compiles things inside functions! So the Julia timing was interpreted rather than compiled code. Just wrap a function around it,

```
function comp()
    x = 1000.0
    for k in 1:100000000
        x *= 0.9999999
    end
    return x
end
```

I get these times

```
julia> t1 = @time ns(); x = comp(); t2 = @time ns();
```

```
0.045399990739156017
```

```
time (s): 0.148444348
```

versus C++:

```
~/tmp> g++ -O foo.cc -o foo
~/tmp> foo
x=0.04539999
```

```
time (s): 0.149457
```
• mantra: strictly type your types, loosely type your functions.
Parameterized type

type TPoint{T}
  A::T
  B::T
end

julia> p = TPoint{Int32}(2,3)
TPoint{Int32}(2,3)
p.A => 2
```julia
julia> (1+2)::AbstractFloat
ERROR: TypeError: typeassert: expected AbstractFloat, got Int64

julia> (1+2)::Int
3

julia> function foo()
    x::Int8 = 100
    x
end
foo (generic function with 1 method)

julia> foo()
100

julia> typeof(ans)
Int8
```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end

primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8   <: Signed  8 end
primitive type UInt8  <: Unsigned 8 end
primitive type Int16  <: Signed  16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32  <: Signed  32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64  <: Signed  64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end
multiple dispatch

using Base
import Base. *

julia> *
* (generic function with 149 methods)

julia> function *(a::String, n::Integer)
    accum = ""
    for i = 1:n
        accum = accum + a  # after defining +
    end
    return accum
end
* (generic function with 150 methods)

julia> "abc" * 4
"abcabcabcabc"
"abcabcabcabc"
o-o versus functional; multiple dispatch

- O-O: state acts like globals to all (inherited) functions of a class


- Arbitrarily asymmetric: dispatch only on the type of the first arg: `a.mul(b)`

- Julia: dispatch on all types, avoid state wherever possible
Composition over inheritance (or composite reuse principle) in object-oriented programming is the principle that classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class. This is an often-stated principle of OOP, such as in the influential book *Design Patterns*.
Type magic

- cudaarray
- symbolic for free
type magic: convert numeric code to symbolic for free
The ODE solvers for Julia are in the package **DifferentialEquations.jl**. Let’s solve the linear ODE:

\[
\frac{dy}{dt} = 2y
\]

with an initial condition which is a symbolic variable. Following the tutorial, let’s swap out the numbers for symbolic expressions. To do this, we simply make the problem type and solve it:

```julia
using DifferentialEquations, SymEngine
y0 = symbols(:y0)
u0 = y0
f = (t,y) -> 2y
prob = ODEProblem(f,u0,(0.0,1.0))
sol = solve(prob,RK4(),dt=1/10)
println(sol)
# SymEngine.Basic[y0,1.2214*y0,1.49181796*y0,1.822106456344*y0,2.2255282577856*y0,2.7182511386618*y0,3.3110475557571*y0,3.9187584072163*y0,4.5323605636419*y0,5.1508976021755*y0,5.7653686268422*y0]
```
Type magic: example: verify a trace identity
For deep learning

- tensor flow: guessing 50+ person years. 100s kloc, c++, cuda, python

- knet: <1 person year

- Fully understandable (tiny source, all in julia), comparable performance (even on gpu)

“Knet: deep learning with 100 lines of julia”:

<table>
<thead>
<tr>
<th>model</th>
<th>dataset</th>
<th>epochs</th>
<th>batch</th>
<th>Knet</th>
<th>Theano</th>
<th>Torch</th>
<th>Caffe</th>
<th>TFlow</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinReg</td>
<td>Housing</td>
<td>10K</td>
<td>506</td>
<td>2.85</td>
<td>1.88</td>
<td>2.66</td>
<td>2.37</td>
<td>5.92</td>
</tr>
<tr>
<td>Softmax</td>
<td>MNIST</td>
<td>10</td>
<td>100</td>
<td>2.35</td>
<td>1.40</td>
<td>2.88</td>
<td>2.82</td>
<td>5.57</td>
</tr>
<tr>
<td>MLP</td>
<td>MNIST</td>
<td>10</td>
<td>100</td>
<td>3.68</td>
<td>2.31</td>
<td>4.03</td>
<td>3.75</td>
<td>6.94</td>
</tr>
<tr>
<td>LeNet</td>
<td>MNIST</td>
<td>1</td>
<td>100</td>
<td>3.59</td>
<td>3.03</td>
<td>1.69</td>
<td>3.54</td>
<td>8.77</td>
</tr>
<tr>
<td>CharLM</td>
<td>Hiawatha</td>
<td>1</td>
<td>128</td>
<td>2.25</td>
<td>4.57</td>
<td>2.23</td>
<td>–</td>
<td>2.86</td>
</tr>
</tbody>
</table>
Metaprogramming

• the only feature a language really needs

• metaprogramming trumps o-o.

  • no O-O? metaprogram it in an afternoon (example: Paul Graham book)

  • O-O but no metaprogramming? oh well
Paul Graham example

- inheritance in 6 lines of code
- extend to before/after methods, method combination, appropriate syntax
- … in an afternoon

(defmacro defmeth (((name &optional (type :primary)) obj parms &body body)
  (let ((gobj (gensym)))
    'let ((,gobj ,obj))
      (defprop ,name t)
      (unless (meth-p (gethash 'name ,gobj))
        (setf (gethash 'name ,gobj) (make-meth)))
      (setf ,(symb 'meth- type) (gethash 'name ,gobj)
        ,(build-meth name type obj parms body))))))

(defun build-meth (name type gobj parms body)
  (let ((,gargs (gensym))
    '#'(lambda (&rest ,gargs)
      (labels
        ((call-next ()
          ,(if (or (eq type :primary)
              (eq type :around))
            '(cnm ,gobj ,'name (cdr ,gargs) ,type)
            'error "Illegal call-next."))
        (next-p ()
          ,(case type
            (:around
              '(or (rget ,gobj ,'name :around 1)
                (rget ,gobj ,'name :primary))
              (:primary
                '(rget ,gobj ,'name :primary 1)
                (t nil))))
          (apply '#'(lambda ,parsms ,@body) ,gargs))))))

(defun cnm (obj name args type)
  (case type
    (:around (let ((ar (rget obj name :around 1)))
      (if ar
        (apply ar obj args)
        (apply named-advice args))))
    (:primary (rget obj name :primary 1))
    (:around (rget obj name :primary 1) (t nil))))
Because `@def` works at compile-time, there is no cost associated with this. Similar metaprogramming can be used to build an "inheritance feature" for Julia. One package which does this is `ConcreteAbstractions.jl` which allows you to add fields to abstract types and make the child types inherit the fields:

```julia
# The abstract type
@base type AbstractFoo{T}
    a
    b::Int
    c::T
    d::Vector{T}
end

# Inheritance
@extend type Foo <: AbstractFoo
    e::T
end
```

where the `@extend` macro generates the type-definition:

```julia
type Foo{T} <: AbstractFoo
    a
    b::Int
    c::T
    d::Vector{T}
    e::T
end
```

But it's just a package? Well, that's the beauty of Julia. Most of Julia is written in Julia, and Julia code is first class and performant (here, this is all at compile-time, so again runtime is not affected
metaprogramming

julia> ex = :(a + b)
:(a + b)

julia> typeof(ex)
Expr

julia> ex.head
:call

julia> ex.args
3-element Array{Any,1}:
  :+
  :a
  :b
metaprogramming

julia> ex = :(a + b)
:(a + b)

julia> ex.\text{args}[2], ex.\text{args}[3] = ex.\text{args}[3], ex.\text{args}[2]
(:b,:a)

julia> ex.\text{args}[1] = :*
:*

julia> ex
:(b * a)
metaprogramming no more copy/paste/bug

Typical example: need nearly parallel code for data structure with .X, .Y, .Z fields
Loop over X,Y,Z fields, generate 3 functions at compile time
each of which sums or averages one of the coordinates.

type Point{T <: Number}
    X::T
    Y::T
    Z::T
end

Pts = Array{Point{Float32}}(3)
Pts[1] = Point{Float32}(1,2,3)  # etc

for (name,field) in ((:sumX, :X), (:sumY, :Y))
    @eval begin
        function $name(ptarr)  # note $name
            thesum = 0
            for i = 1:length(ptarr)
                println(ptarr[i].$fieldname)
                thesum += ptarr[i].$fieldname  # note $fieldname
            end
            thesum
        end
    end
end
Because `@def` works at compile-time, there is no cost associated with this. Similar metaprogramming can be used to build an "inheritance feature" for Julia. One package which does this is `ConcreteAbstractions.jl` which allows you to add fields to abstract types and make the child types inherit the fields:

```julia
# The abstract type
@base type AbstractFoo{T}
  a
  b::Int
  c::T
  d::Vector{T}
end

# Inheritance
@extend type Foo <: AbstractFoo
  e::T
end

type Foo{T} <: AbstractFoo
  a
  b::Int
  c::T
  d::Vector{T}
  e::T
end
```
Other stuff

- Jupiter
- make an executable
- unicode variables
- `@code_warntype (type stability)`
- `@inbounds` for `i=1:n` ...
π in Julia

(Simon Byrne)

Like most technical languages, Julia provides a variable constant for π. However Julia’s handling is a bit special.

```julia
pi
π = 3.1415926535897...
```

It can also be accessed via the unicode symbol (you can get it at the REPL or in a notebook via the TeX completion \\pi followed by a tab)

```julia
π
\π = 3.1415926535897...
```

You’ll notice that it doesn’t print like an ordinary floating point number: that’s because it isn’t one.

```julia
typeof(pi)
Irrational{::π}
```
\[ \pi \text{ via inline assembly instructions} \]

*(Simon Byrne)*

Julia provides a very low-level `llvmcall` interface, which allows the user to directly write **LLVM intermediate representation**, including the use of inline assembly. The following snippet calls the `fldpi` instruction ("floating point load \(\pi\)") which loads the constant \(\pi\) onto the floating point register stack (this works only on x86 and x86_64 architectures)

```julia
function asm_pi()
    Base.llvmcall(
        """ %pi = call double asm "fldpi", "={st}" ()
        ret double %pi"
    ,
    Float64, Tuple{})
end

asm_pi (generic function with 1 method)

asm_pi()

3.141592653589793