two_language_problem

December 5, 2020

#
Introduction to Julia: Solving Two-Language Problem

1 1. Two-Language Problem

- Dynamic languages (e.g., Python, R, Matlab) tend to be slow
- Lower-level, compiled languages (e.g., C, C++, Fortran) are very fast, but are more time-consuming to write and debug

1.1 The Standard Approach - Write core algorithms in lower-level language - Then wrap that in higher-level language using some interface language/package (e.g., Cython, Rcpp)

1.1 1.2 What's the Problem?

1. Higher barrier to entry for package authors
2. Package authors must know (at least) two languages
3. Creates a sharp divide between package authors and package users

#
Meet Julia!

2 2. Julia Language

1. Dynamic technical computing lanugage
2. Multi-paradigm (functional, imperative, "OO"-ish)
3. Just-in-time compiled using LLVM
4. Under active development (current stable version: 1.5.3)
5. Fast!
6. Fun!

2.1 2.1 History of Julia

- Started in 2009 as PhD thesis of Jeff Bezanson at MIT
- Jeff has been collaborating with Stefan Karpinski and Viral Shah
- First official release in 2012
- Influenced by: C, Python, R, Ruby, Lisp
- Specifically designed to be as fast as C and as expressive as Python or Ruby

2.2 2.2 Julia Basics

- Similar to Python and R in that:
  - Very flexible
  - Has elements of imperative, functional, and object-oriented programming
  - Functions are first-class citizens
    - (e.g., pass as arguments, return from other functions)

2.2.1 2.2.1 Julia Basics (cont.)

- Differs from Python and R in that:
  - Compiled, not interpretted
  - Not object-oriented in the classical sense
  - No classes with private data and private methods
  - No concept of inheritence
  - Designed with parallelism in mind
  - Multiple dispatch
  - Excellent for generic programming
  - Metaprogramming

2.3 2.3 Base Language Data Types

1. Primitives

- Any numeric type you can imagine: `Int64`, `Float64`, `BigInt`, `Complex`, `Irrational`, `Rational`
- Many abstract types: `Any`, `Real`, `Number`, `Integer`
- `String` and `Chars`

2. Container Types

- `Array` (vectors, matrices, *N*-dimensional arrays)
- `Set`
- `Dict`
- `Tuple` and `NamedTuple`

3. Composite Types

- `struct`

In [3]:
```
# Simple function that computes Fibonacci numbers

function fib(n)
    nums = ones(Int, n)
    for i = 3:n
        nums[i] = nums[i - 1] + nums[i - 2]
    end
    return nums[n]
end
```

```
Out[3]: fib (generic function with 1 method)
```

```
In [4]: fib(50)
```

```
Out[4]: 12586269025
```

#
Julia is Fast!

```
In [5]: function qsort(a, lo, hi)
            i = lo
            j = hi
            while i < hi
                pivot = a[div(lo + hi, 2)]
                while i <= j
                    while a[i] < pivot
                        i += 1
                    end
                    while a[j] > pivot
                        j -= 1
                    end
                    if i <= j
                        a[i], a[j] = a[j], a[i]
                        i += 1
                        j -= 1
                    end
                end
                if lo < j
                    qsort(a, lo, j)
                end
                lo = i
                j = hi
            end
            return a
        end
```

```
Out[5]: qsort (generic function with 1 method)
```

###
R quicksort example

```
In [7]: # quick sort in Julia
        using BenchmarkTools

        n = 100_000
        x = randn(n)

        @time x2 = qsort(x, 1, n);
```

```
  0.010187 seconds
```

2.4 Understanding Julia's Speed

1. Type system makes inferring concrete types easy
2. Julia aggressively specializes run-time types
3. LLVM generates fast native code
4. Very smart people working hard!

#
Parrallelism in Julia

2.5 Parallel Programming in Julia

1. Can use multi-processing or multi-threading
2. Excellent support for SIMD operations
3. `SharedArray` data type in base language
4. `DistributedArray` data structure allows for Hadoop-like distributed computing

Suppose we want to constrain range of values a vector's elements can take to be between -0.5 and 0.5

```
In [8]: function squeeze_range!(x)
            n = length(x)
            for i = 1:n
                if !(-0.5  x[i]  0.5)
                    x[i] = 0.5 * sign(x[i])
                end
            end
        end
```

```
Out[8]: squeeze_range! (generic function with 1 method)
```

```
In [9]: n = 100_000_000
        v = randn(n)

        @time squeeze_range!(v)
```

```
  0.599651 seconds (20.43 k allocations: 1.113 MiB)
```

2.5.1 Using Threads for Parallelism

```
In [10]: # NOTE: JULIA_NUM_THREADS env variable is set in .bash_profile
         using Base.Threads

         function thr_squeeze_range!(x)
             n = length(x)
             @threads for i = 1:n
                 if !(-0.5  x[i]  0.5)
                     x[i] = 0.5 * sign(x[i])
                 end
```

```
            end
        end
```

```
Out[10]: thr_squeeze_range! (generic function with 1 method)
```

```
In [12]: n = 100_000_000
         v = randn(n)

         @time thr_squeeze_range!(v)

  0.123301 seconds (36 allocations: 4.281 KiB)
```

#
Integration with other Languages

2.6 Julia and other Languages

Julia has zero-cost interfaces with many technical computing languages:
1. PyCall.jl 2. RCall.jl 3. `ccall()` 4. Cxx.jl 5. Matlab.jl
Introduction to Julia

2.7 Arithmetic

```
In [13]: 3 + 7     # addition

Out[13]: 10

In [14]: 10 - 3    # subtraction

Out[14]: 7

In [15]: 20 * 5    # multiplication

Out[15]: 100

In [16]: 100 / 10 # division

Out[16]: 10.0

In [17]: 10 ^ 2    # exponentiation

Out[17]: 100
```

2.8 Logical Operators

```
In [ ]: false && true   # logical AND

In [19]: false || true   # logical OR

Out[19]: true
```

2.9 Comparisons

```
In [20]: 1 == 1.0 # Equality

Out[20]: true

In [21]: 3 <

Out[21]: true

In [22]: 1 <= 1

Out[22]: true
```

2.10 Assignment

Assignment in Julia is done with the single =. All it does is associates a name (on the left) to a value (on the right).

```
In [ ]: x = 1

In [23]: x = "hello!"

Out[23]: "hello!"
```

2.11 Arrays

Julia has highly efficient multidimensional arrays, both constructed and indexed with square brackets.

```
[item1, item2, ...]
```

```
In [24]: # create array (vector) with []
         squares = [1, 4, 9, 15, 25]

Out[24]: 5-element Array{Int64,1}:
           1
           4
           9
          15
          25

In [25]: squares[1]    # indexing is similar to R

Out[25]: 1

In [26]: squares[1:3] # slicing is same as R

Out[26]: 3-element Array{Int64,1}:
          1
          4
          9
```

2.12 Loops

The syntax for a `for` loop is

```
for *var* in *loop iterable*
    *loop body*
end
```

```
In [27]: for x in 1:3
             println("yo")
         end
```

```
yo
yo
yo
```

```
In [28]: x = 3
         while x < 6
             println(x)
             x += 1
         end
```

```
3
4
5
```

3 if/else

```
In [29]: if 4 > 2
             println("potato!!")
         end
```

```
potato!!
```

```
In [30]: if 3 < 0.4
             println("no")
         else
             println("soup!!")
         end
```

```
soup!!
```

4 Functions in Julia

```
In [31]: function add_one(a)
             # function body
             b = a + 1
             return b
         end
```

```
Out[31]: add_one (generic function with 1 method)
```

```
In [32]: add_one(42)
```

```
Out[32]: 43
```