So let us benchmark now.

Here is the hypergeometric function of a matrix argument:

$$ {}_pF_q^{(\alpha)} \left(\begin{matrix} a_1, \ldots, a_p \\ b_1, \ldots, b_q\end{matrix}; X\right) = \sum_{k=0}^\infty\sum_{\kappa \vdash k} \frac{{(a_1)}_\kappa^{(\alpha)} \cdots {(a_p)}_\kappa^{(\alpha)}} {{(b_1)}_\kappa^{(\alpha)} \cdots {(b_q)}_\kappa^{(\alpha)}} \frac{C_\kappa^{(\alpha)}(X)}{k!}. $$

Well, I will not explain this expression. But observe that this is a sum from $k=0$ to ∞. The algorithm evaluates the partial sums of this series, that is, the sum from $k=0$ to an integer m.

My Haskell library generates a shared library (a DLL) which can be called from R. And one can call Julia from R with the help of the 'XRJulia' package. So we will benchmark the three implementations from R.

Firstly, let's check that they return the same value:

```
library(HypergeoMat)
library(XRJulia)
# source the Julia code
juliaSource("HypergeomPQ09.jl")
# load the Haskell DLL
dll <- "libHypergeom.so"
dyn.load(dll)
.C("HsStart")

a <- c(8, 7, 3)
b <- c(9, 16)
x <- c(0.1, 0.2, 0.3)
alpha <- 2
m <- 5L # `m` is the truncation order

hypergeomPFQ(m, a, b, x, alpha)
# 2.116251
juliaEval("hypergeom(5, [8.0, 7.0, 3.0], [9.0, 16.0], [0.1, 0.2, 0.3],
2.0)")
# 2.116251
.Call("hypergeomR", m, a, b, x, alpha)
# 2.116251
```

Well, the same results. Now, let's run a first series of benchmarks, for $m=5$.

```
library(microbenchmark)
microbenchmark(
```

```
  Rcpp =
    hypergeomPFQ(m, a, b, x, alpha),
  Julia =
    juliaEval("hypergeom(5, [8.0, 7.0, 3.0], [9.0, 16.0], [0.1, 0.2,
0.3], 2.0)"),
  Haskell =
    .Call("hypergeomR", m, a, b, x, alpha),
  times = 10
)

Unit: microseconds
    expr      min        lq       mean    median        uq       max
neval cld
    Rcpp  356.682   623.807   837.7237   827.402  1084.191  1382.500
10  a
   Julia 4052.000 47767.565 44725.3895 48845.156 50597.779 51308.089
10   b
 Haskell  610.852  1136.963  1343.7442  1289.435  1504.323  2650.976
10  a
```

Should we conclude that Rcpp is the winner, and that Julia is slow? That's not sure. Observe that the unit of these durations is the microsecond. Perhaps the call to Julia via `juliaEval` is time-consuming, as well as the call to the Haskell DLL via `.Call`.

So let us try with \(m=40\) now.

```
m <- 40L
microbenchmark(
  Rcpp =
    hypergeomPFQ(m, a, b, x, alpha),
  Julia =
    juliaEval("hypergeom(40, [8.0, 7.0, 3.0], [9.0, 16.0], [0.1, 0.2,
0.3], 2.0)"),
  Haskell =
    .Call("hypergeomR", m, a, b, x, alpha),
  times = 10
)

Unit: seconds
    expr       min        lq      mean    median        uq      max
neval cld
    Rcpp 25.547556 25.924749 26.130888 26.185776 26.354177 26.47846
10   c
   Julia 18.959032 19.088749 19.191394 19.173662 19.291175 19.62415
10  b
 Haskell  6.642601  6.653627  6.736082  6.735448  6.760926  6.94283
10 a
```

This time, the unit is the second. Haskell is clearly the winner, followed by Julia.

I'm using Julia 1.2.0, and I have been told that there is a great

improvement of performance in Julia 1.5.0, the latest version. I'll try with Julia 1.5.0 and then I will update this post to show whether there is a gain of speed.

One should not conclude from this experiment that Haskell *always* beats C++. That depends on the algorithm we benchmark. This one intensively uses recursion, and perhaps Haskell is strong when dealing with recursion.

Don't forget:

```
dyn.unload(dll)
```

Update: Julia 1.5 is amazing

Now I upgraded Julia to the latest version, 1.5.2. The results are amazing:

```
Unit: seconds
    expr       min        lq      mean    median        uq       max
neval cld
    Rcpp 23.464676 24.392115 24.860484 24.823062 25.013047 27.437176
10   c
   Julia  2.806364  2.852674  3.101521  2.973963  3.363618  3.897855
10 a
 Haskell  6.912441  7.459939  7.648012  7.674404  7.798719  8.322777
10  b
```

19 seconds for Julia 1.2.0 and 3 seconds for Julia 1.5.2! It beats Haskell.