On Julia

Ben Klemens

9 December 2013

A colleague[1] talked me into trying Julia[2]. My resistance was based on two immediate objections.

First, do we really need a new language?

Second, do we really need a new domain-specific language? DSLs are usually reserved for very specific problems, like the regular expression sublanguage used to describe text expressions with regularities; the XPath sublanguage used to find bits of XML; SQL for talking to tables. A good DSL knows its place. Even if your flavor of SQL is Turing Complete, using it to write programs that do more than subset and join tables is little more than an interesting exercise.

Conversely, now and then you find a DSL that went too far. Matlab and R were originally built around certain types of workflow, explicitly described in early documentation (in the case of S and R, see Chambers's book; for Matlab see this discussion of Matlab's early history[3]). Within the field, people try to apply the DSL to expanding fields of use, and shoehorning happens. [By the way, shoehorns are great. Using one is incredibly smooth and just feels good. Yet the word *shoehorning* has negative connotations.] Across fields, having multiple DSLs expanding their scope is the setup for little battles: will the language used by traditional statisticians expand its empire into fields where everybody is using Matlab? Will the language electrical engineers know and rely on be able to win over statisticians? The result is Balkanization and isolation for no reason.

After all, here is the procedure for building the solution to any problem in code:

- Break down your problem into subproblems.
- Find the existing code that solved each subproblem.
- Duct tape them together to solve the main problem.

If all the favorite electrical engineering procedures are in one language and all the regression procedures are in another, then findability and composability suffer, and people waste time rewriting the same thing in each subfield.

So the first reason I'm trying Julia is that my colleague convinced me that Julia has pretentions toward a general-purpose language that happens to be good with numeric code. The culture behind Julia, by my reading, pushes reuse and reinvention and is written from the ground up to make that easy (as the manual's introduction[4] explicitly

[1] `http://www.tokle.us/`

[2] `http://julialang.org`

[3] `http://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html`

[4] `http://docs.julialang.org/en/latest/manual/introduction/`

states, as of this writing). Julia impresses me as an effort to provide a common front-end, as inclusive rather than exclusive.

One of my motivations in writing Apophenia in C was that just about every language has a hook to call C code, so if we have a reliable library of statistics functions in C, then people writing new systems could link to that rather than rewriting everything yet again. Some systems make this linking easier than others. With Julia, it is *really easy* to call C code. Apophenia has two types to speak of: `apop_data` and `apop_model`. Build conduits for those two types, and everything else falls into place. With very little knowledge of Julia, writing a working wrapper for `apop_data` took me about half an hour. I spent an hour or two doing that *this should work why doesn't it work oh wait lemme try this* sort of debugging with the `apop_model`, but throughout I had the sense that I was making progress and it was just a question of having 100% of the details right. Next time, I'll demo the results.

It was a downright wonderful experience compared to writing wrappers for another DSL, and even producing a Python wrapper using Swig, the general-purpose wrapper generator (which has a couple of undocumented eccentricities that I ran into head-on). It didn't assume that C code consists exclusively of loose arrays of `int`, `float`, and `char*` (even though that's how C textbooks from the 1980s characterize the language's use). The documentation was clear and immediately available, and at no point did it hint that if I were a better person I wouldn't be using C at all and would be doing everything in Julia.

Julia's package management system is named *Git*. A central list keeps the location of the repositories, which are in a predefined format[5]. Once again, reuse saves the day. I've blogged on the Balkanization of package managers (entry #044) before, and because Git is neutral ground, I have some hope that it could perhaps become the infrastructure for a cross-language package system. We'll see where Julia goes and whether it bureaucratizes with age, but at the moment, you don't have to go through the oppposite-of-open-source process of satisfying a committee[6] to have a package distributed widely.

As for the details of the language, like the well-thought-out type system, the Matlab-like block statements which everybody (myself included) would rather see changed to Python-like block statements, and other such minutiæ, I'll refrain from comment. Also, I still like being able to exclaim, when I have a new problem, *Do it to Julia!*, though I expect that novelty will wear off eventually.

As a postscript, the thing that makes all of this possible is LLVM[7], a language-agnostic compiler for which it is relatively easy to write new front-ends for new languages. LLVM is already taking over the world[8], and its ability to cross languages also gives me hope for the future.

Next time: the aforementioned draft of a Julia wrapper, and a demo re-running an earlier sample program in Julia.

---

[5] `http://docs.julialang.org/en/release-0.1/manual/packages/#contributing-a-new-julia-package`

[6] `https://stat.ethz.ch/pipermail/r-sig-mixed-models/attachments/20131209/ec911711/attachment.pl`

[7] `http://llvm.org/`

[8] `http://acko.net/blog/on-asmjs/`