An Empirical Analysis of Julia Usage

Isaac Virshup

February 24, 2016

0.1 Introduction

People care a lot about the programming languages they use. Reasons for language preference range from the technical to preferential. While technical features of a language can be proved, measuring usability requires an entirely different – and often ignored – approach considering users of the language. Studying programmers allows for insight on ease of use, common practices (patterns), and directions for language improvement. I propose an exploratory analysis of the Julia ecosystem at large to examine how Julia is being used.

0.2 Why do research into how a programming language is used?

It's hard to write good software, so patterns and abstraction (language *features*) are created to make it easier. Measures of these features effectiveness used by language designers often don't meaningfully consider their usage by developers. Technical approaches can be used to say if a feature solves a problem faster, or in particular cases. When this approach is applicable, that's great. Using a formal solution for a formal problem makes sense. But this approach does nothing to say if a feature is usable [5].

Measuring usability requires studying feature usage – including developers in the system being studied. There's a history of language designers making unsubstantiated claims a language having "desirable qualities" – like naturalness, readability, practicality, and efficiency [7]. These qualities are vague, qualitative, and generally hard to measure. Still, it is important to investigate the qualitative side of a language. How do you know if your feature makes solving problems easier without investigating it's usage? Benefits of considering what programmers do with a language extend beyond syntactic improvement. Early work considering language usage [6] emphasized it's value for guiding compiler optimization and design.

Interest in investigating use and adoption of programming languages and features has been going through a resurgence recently, under the name of empirical software engineering [5]. This field looks to come up with metrics relevant to the usefulness of language and software features, as to help guide their development. Methods used look to exploit progress made in the quantitative study of sociological systems. For example [8] suggests the empirical software engineering community look to the fields of public health and historical linguistics for methodological design.

Why Julia

Being currently under development, Julia provides a great opportunity to apply methods from empirical software analysis. Evidenced by activity on "decision" issues of `JuliaLang/julia`[1], it's in a flexible state. Quantitative information on what users are doing and how they are reacting to previous work could be extremely useful for language designers. At the very least, it can suggest the impact of their changes and inform them as to how features are used.

Development patterns in Julia itself make it a good subject for study. Importance placed by designers on features like introspection and static analysis means a solid suite of tools already exists. In addition, since development is highly GitHub-centric archival source code and metadata from package and language development is easily available through a single API.

0.3 Proposal

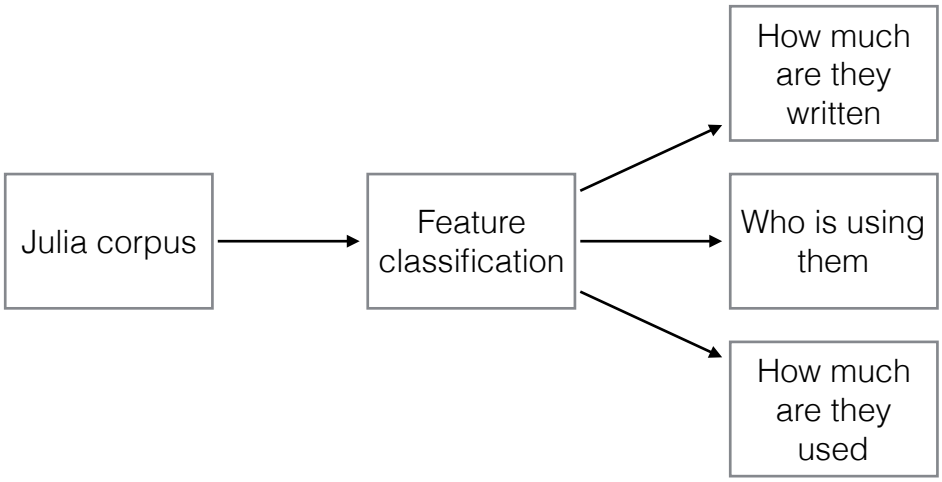

Figure 0.1: Project layout. A corpus of Julia code will be constructed. Classifiers will then be applied to the corpus to find the usage of features of the Julia language. This usage will then be analyzed by statistics of counts, analyzing who used different features, and how much each of these features is actually used at runtime.

Research questions

- How much do Julia's users use it's unique?
- Does usage of Julia vary by community?

[1] https://github.com/JuliaLang/julia/labels/decision

- Can information immediately relevant to the language design process be extracted from language usage?

Approach

Establishing a corpus

A corpus should be fairly easy to establish. The built in `METADATA.jl` package contains access to a number of community developed packages, whose contribution data I've already been able to access[2]. While a study of feature usage in [1] looked just `Base`, I think it will be important to broaden the scope of the analysis as `Base` may be a biased sample. Investigation will be needed to determine if this is sufficient for analysis, but it's a good starting point.

Classifiers

I'll have to create methods for matching the features I'd like to look at. Doing a dynamic analysis (as was done in [1]) gives me access to all of Julia's introspection and metaprogramming features, but raises issues with importing modules into the scope I'm performing the analysis in. `JuliaParser.jl` provides the tools necessary to do a static analysis of the code by generating ASTs. A similar approach was successful at large scales in studies of Java feature adoption [3]. An exciting, but likely difficult to implement method is the *ab initio* detection of design patterns in Julia code [11].

A preliminary list of features I would like to look into:

- Strength of typing
- User created types, in all their variations (parametric, abstract, etc.)
- Parallel features
- Multiple dispatch
- Macros/metaprogramming features
- Calls to other languages, which languages they are calling to

Analysis

How much are features being used An obvious and early analysis will be to look at how much the various features are being used/written. I intend to follow an abstracted version of the methodology used by [6], finding usage counts of various features and try to qualify how they are used. A similar (if more sophisticated) kind of analysis was applied to multiple dispatch in Julia `Base` by [1] based on methods taken from [9].

Who's using what features Are there trends in who uses Julia features? Studies of other languages have found divides by community, for example [6] found that Stanford programmers usage of FORTRAN features differed significantly that by Lockheed developers. In addition, are features broadly used in the Julia community, or is there specialization. In studies of Java it's been found that typically only a few users are responsible for the majority of implementations of new features [10] [3].

[2] Of 886 sampled Julia GitHub projects, there were 1154 unique contributing developers - 442 of whom contributed to `JuliaLang/julia`. 82715 out of 121212 total contributions were to `JuliaLang/julia`.

How much are features being called Because a feature isn't being directly used by a programmer doesn't mean it's not integral to their code. They could be using an abstraction defined in another package. For example, calling `ComputeFramework.jl` instead of the built in parallel features.. To get a full picture of Julia's usage this will need to be accounted for.

Methods for this approach range from taking profiles of running programs [6], to performing impact analysis [4] [2]. Barriers to implementation exist with both methodologies. Profiling will require finding a program that runs, and having some assurance it is "typical". Potential sources include `Gists` and tests for packages. Most large scale examples of impact analysis are done on static, strongly-typed languages and are reliant on those features.

0.4 Conclusion

Julia is an excellent environment to apply big-data methods with empirical software analysis. Fitting with it's research ethos, this will take advantage of an opportunity to see how a quantitative analysis of language use can inform language development. Julia is well primed for this work, with an emphasis on introspection features and development structured around open-source.

Bibliography

[1] Jeff Bezanson, Jiahao Chen, Stefan Karpinski, Viral Shah, and Alan Edelman. Array Operators Using Multiple Dispatch: A design methodology for array implementations in dynamic languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 56. ACM, 2014. 3

[2] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings - International Conference on Software Engineering*, pages 419–429. University of California, Riverside, United States, IEEE, July 2012. 4

[3] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*, pages 779–790. ACM, 2014. 3

[4] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *th International Conference on Software Engineering ICSE*, pages 430–440. IEEE, IEEE, 2012. 4

[5] Stefan Hanenberg. Faith, Hope, and Love: An Essay on Software Science's Neglect of Human Factors. *SIGPLAN Not.*, 45(10):933–946, October 2010. 1

[6] Donald E Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, April 1971. 1, 3, 4

[7] Shane Markstrum. Staking claims: a history of programming language design claims and evidence: a positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, page 7. ACM, 2010. 1

[8] Leo A Meyerovich and Ariel S Rabkin. Socio-PLT: Principles for programming language adoption. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 39–54. ACM, 2012. 1

[9] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Acm sigplan notices*, pages 563–582. ACM, 2008. 3

[10] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 3–12. ACM, 2011. 3

[11] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, 2006. 3