What Questions Do Developers Ask About Julia?

Dibyendu Brinto Bose
Reve System
Dhaka, Bangladesh
brintodibyendu@gmail.com

Gerald C. Gannod
Tennessee Tech University
Cookeville, Tennessee, USA
jgannod@tntech.edu

Akond Rahman
Tennessee Tech University
Cookeville, Tennessee, USA
arahman@tntech.edu

Kaitlyn Cottrell
Tennessee Tech University
Cookeville, Tennessee, USA
kmcottrell42@tntech.edu

ABSTRACT

The programming language Julia is designed to solve the 'two language problem', where developers who write scientific software can achieve desired performance, without sacrificing productivity. Since its inception in 2012, developers who have been using other programming languages have transitioned to Julia. A systematic investigation of the questions that developers ask about Julia can help in understanding the challenges that developers face while using Julia. Such understanding can be helpful (i) for toolsmiths who can construct tools so that developers can maximize their experience of using Julia, and (ii) for Julia language maintainers with empirical evidence on areas to improve the language as well as the Julia ecosystem. We conduct an empirical study with 3,093 Stack Overflow posts where we identify 13 categories of questions related to Julia-based software development. We observe developers to ask about a diverse set of topics, such as `GC`, Julia's garbage collector, `JuMP`, a domain-specific language constructed using Julia, and `symbols`, a metaprogramming utility in Julia. Based on our emerging results, we recommend enhancing support for developers with Julia-based tools and techniques for cross language transfer, type-related assistance, and package resolution.

CCS CONCEPTS

• **Software and its engineering** → *Frameworks.*

KEYWORDS

Challenges, Empirical Study, Julia, Stack Overflow

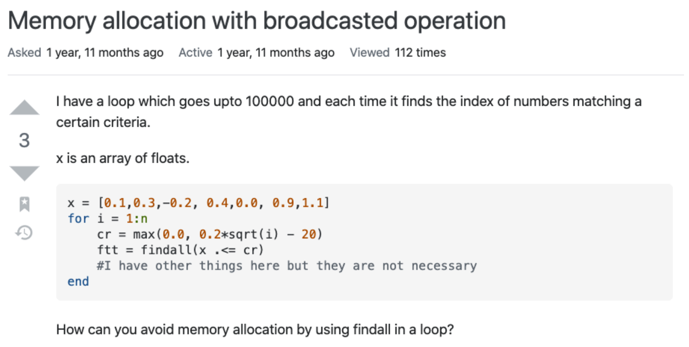
ACM Reference Format:
Dibyendu Brinto Bose, Gerald C. Gannod, Akond Rahman, and Kaitlyn Cottrell. 2022. What Questions Do Developers Ask About Julia?. In *2022 ACM Southeast Conference (ACMSE 2022), April 18–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3476883.3520205

1 INTRODUCTION

Typically developers who develop scientific software, rely on scripting languages, such as Python [10]. While these scripting languages help in developer productivity, can hinder program execution speed, as these languages do not provide a predictable mapping between the program and the hardware [10]. As a result, developers have to migrate their software source code base to C or Fortran to achieve desired program execution speed. Such migration usually leads to improved program execution speed for the software project, but yield development and maintenance overhead [10]. The programming language Julia is designed so that developers involved in scientific software development do not have to transition from one language to another. Creators of Julia designed the language to solve the 'two language problem', which allows developers to achieve desired performance, without sacrificing productivity [2].

According to a survey of Stack Overflow (SO) users in 2020, Julia is considered as one of the "*top 10 most loved programming languages*" by practitioners [5]. We observe Julia being used in research and product development as well. For example, Julia was used in Celeste [1], a software used in astronomy research. Celeste was used to load 178 terabytes of astronomical image data to produce a catalog of 188 million astronomical objects in 14.6 minutes, yielding a performance improvement by a factor of 1,000, compared to prior implementation [1]. As of Jan 2021, Julia has been downloaded 24,205,141 times [4].

The above-mentioned discussion shows that Julia is an emerging programming language, as developers involved in scientific software development are switching from scripting languages, such as Python to Julia [6]. Despite finding beneficial in software projects, developers face challenges in using Julia as expressed in forms of questions that are posted on question and answer websites, such as Stack Overflow (SO). Let us consider Figure 1 in this regard. We observe a developer to ask about how memory is allocated when performing array broadcasting in Julia. Broadcasting refers to the feature of performing element-by-element operations on arrays of different sizes, e.g., adding a vector to each column of a matrix [3]. Evidence presented in Figure 1 demonstrates that while using Julia, developers encounter difficulties, and seek help from the SO community. As an emerging community, developers who use Julia can benefit from an empirical study that systematically investigates the questions that developers ask about Julia. Such study can help the software engineering research community understand the challenges that developers face while using Julia, which in turn can



Memory allocation with broadcasted operation

Asked 1 year, 11 months ago Active 1 year, 11 months ago Viewed 112 times

3

I have a loop which goes upto 100000 and each time it finds the index of numbers matching a certain criteria.

x is an array of floats.

```
x = [0.1,0.3,-0.2, 0.4,0.0, 0.9,1.1]
for i = 1:n
    cr = max(0.0, 0.2*sqrt(i) - 20)
    ftt = findall(x .<= cr)
    #I have other things here but they are not necessary
end
```

How can you avoid memory allocation by using findall in a loop?

Figure 1: An Example Julia-related SO Question [26]

yield derivation of tools and practices so that developers can maximize their experience in using Julia. Furthermore, Julia language maintainers can use the obtained empirical insights to improve the language as well as the Julia ecosystem.

We answer the following research question: **RQ:** *What questions do developers ask about when writing Julia programs?*

Our contribution is *a list of categories related to questions that developers ask about when using Julia.*

We organize rest of the paper as follows: we discuss background and related work in Section 2. We describe our empirical study in Section 3. We discuss our findings and limitations respectively, in Sections 4 and 5. We conclude our paper in Section 6.

2 BACKGROUND AND RELATED WORK

In this section, we provide background information and discuss research relevant to our paper:

2.1 Background

Julia is perceived to solve the 'two language problem' [10], which refers to the phenomenon of practitioners having to switch to a programming language that is harder to use in order to achieve better performance. For example, writing programs in Python can be relatively easy for practitioners because of its scripting nature. However, Python programs' execution time may not be as fast as C programs. Rapid program execution can be desirable when applying complex computational activities on large-scale datasets, e.g., as done by the Celeste project [1].

Julia programs can call low-level functions from the C runtime. Julia programs also take advantage of just-in-time (JIT) compilation, which is the process of compiling lines of code sequentially as they are seen, instead of compiling all lines beforehand. With the use of JIT compilation, Julia is perceived to be useful in developing computationally efficient programs. Julia programs are also compiled into an intermediate representation of bytecode, which allows portability between different computer architectures.

We provide an annotated example of a Julia program in Figure 2. Dedicated code elements, such as `include` and `println` respectively, are used to specify dependencies and redirect program output to the console. A collection of Julia programs is referred to as a package. Functions in Julia are defined using the `function` keyword. Julia allows the return of one or multiple values without explicitly specifying the `return` keyword, as long as the values that need to be returned reside on the last line in the function body. For example, in Figure 2, the function `mul_and_add` performs two mathematical operations, multiplication and addition, and returns the results of those two operations by using the statement `m, a`, which is the last sentence in the function body. String interpolation is also possible using the `$` character.

```
# Enable pre-compilation
__precompile__()
# Create example module
module Example
 # Include a package dependency
 include("Sample.jl")
 # Simple hello world standard output
 println("Hello World")
 # Function to multiply and add two values
 function mul_and_add(a, b)
    m = a*b
    a = a+b
    m, a
 end
 r1, r2 = mul_and_add(3, 4) # result:  12, 7
 # Create macros with "macro" keyword
 macro assert_str(s)
   return :($s
    ? nothing
    : throw(AssertionError($(string(s)))))
 end
 @assert_str 1 == 1.0 # result:  nothing
 @assert_str 1 == 0
 # result:  ERROR: AssertionError:  1 == 0
end
```

Figure 2: An Annotated Example of a Julia Program

2.2 Related Work

Since its inception in 2012, Julia has garnered interest amongst researchers. Quality issues in Julia programs have been investigated, e.g., Paulding and Feldt [20] applied random testing to find faults in 9 Julia-provided functions. Churavy [11] constructed and evaluated a debugging tool called 'Cthulhu' that uses static and dynamic analysis to help developers find bugs in array abstractions. Nardelli et al. [29] used formal specification to verify the correctness of Julia-related subtypes. Productivity and performance issues have also been investigated: Gibson [15] reported that Julia has multiple benefits over existing programming languages with respect to graphic rendering capabilities, user experience, and program execution time. Januszek et al. [17] compared the performance of five programming languages for algorithms with $O(n^3)$ time complexity, and observed superior computational efficiency for Julia programs compared to that of Wolfram, R, Python, and C# programs. For parallel programming, Gmys et al. [16] found Julia to be better than Python with respect to performance, and better than C programs with respect to productivity. Sells [23] used an open-source, industry-standard, missile and rocket simulation software called 'Mini-Rocket' to assess and benchmark productivity metrics of Julia against Python, Java, and C++. Their [23] results

Table 1: Selection of Julia-related SO Questions

Initial question count	18,597,996
Criterion-1 (Questions tagged as 'julia')	6,361
Criterion-2 (Questions with at least one answer)	5,678
Criterion-3 (Questions with accepted answers)	4,355
Criterion-4 (Questions with score > 0)	4,355
Criterion-5 (Questions with > 0 views)	4,355
Criterion-6 (Questions with code snippets)	3,093
Final question count	3,093

showed that Julia required far less lines of code than the other three languages and was second best only to Python in terms of 'ease-of-coding' productivity. Dogaru et al. [13] observed that Julia's base JIT implementation was 157.5 times faster than raw Python code and 3.09 times faster than JIT-assisted Python code. Farhana et al. [14]'s paper on synthesizing challenges related to Julia program performance is the closest to our paper in spirit. However, they [14] did not investigate the programming-related challenges for Julia.

The above-mentioned discussion highlights research that have investigated quality and performance comparison for Julia programs. We observe a lack of research related to challenges expressed in forms of SO questions that developers ask about Julia. We address this research gap in our paper.

3 EMPIRICAL STUDY

In this section, we provide the methodology and results for **RQ: What questions do developers ask about when writing Julia programs?**

3.1 Methodology

Dataset: We mine SO questions by using the SOTorrent dataset [9], which we downloaded on April 15, 2021. According to prior work [21], SO datasets suffer from quality issues. Similar to prior research [21], we apply a filtering criteria to improve the quality of the downloaded data, which is summarized in Table 1. Altogether, we collect 3,093 SO questions.

Open coding: We apply a qualitative analysis technique called open coding [22] on the collected 3,093 SO questions. In open coding, a rater observes and synthesizes patterns within unstructured text [22]. As part of our open coding process, *first*, the rater reads each SO question title, description, comments, and answers to obtain raw text, which is merged into codes. *Second*, the rater merges the codes based on similarities to derive categories.

The first author derived the categories. The derivation of categories is susceptible to bias. We verify the first author's rating by allocating another rater, who is the last author of the paper. The last author applied closed coding [12] on a randomly selected set of 500 SO questions. For each of the 500 SO questions, the last author examined if the question maps to any of the categories identified by the first author. The first and last authors respectively, have 1 and 8 years of experience in software engineering. Upon completion of the inspection process, we calculate Krippendorff's α [18] to quantify agreement, similar to prior work in software engineering [7]. The Krippendorff's α is 0.93, indicating 'acceptable' agreement [18].

3.2 Results

We identify 13 question categories that developers ask about Julia. We describe the categories below where we present each category sorted based on the count of questions that belong to each category. Description of each category includes examples that are presented in the ($_{QID}$) format, where *QID* is the ID of the SO question. The count of questions that map to each challenge is enclosed within parenthesis. For example, 790 of the studied 3,093 questions belong to the category 'Visualization'.

1. **Visualization (790)** This category consists of questions that are related to generating visualizations, such as bar plots and box plots. For example, in a SO question ($_{29310646}$), a developer asks about generating a bar plot using a specific color.

2. **Array Manipulation (495)** This category consists of questions related to array operations in Julia. Example of a SO question ($_{30699805}$) related to array manipulation is how to conduct element-by-element comparison in Julia arrays.

3. **Package Resolution (435)**: This category consists of questions that describe installation, management, and usage of Julia packages that are required to develop Julia-based software projects. The category includes questions related to installing, uninstalling, using, and resolving unmet package dependencies. Example of a question related to package resolution was observed for the `TimeSeries` package in a SO question ($_{31786795}$). The developer used an empty string (`""`) in the header list for a comma-separated value (CSV) file, which triggered a program crash for the `Readtimearray` function available as part of the `TimeSeries` package. The program crash was fixed by removing the empty string from the header list.

4. **Program Execution Speed (279)**: This category consists of questions that are related to program execution speed of Julia programs. This category includes questions related to program execution time, memory allocation, and parallel programming of Julia. In a SO question ($_{31656858}$), a developer observed performance decrease while using parallelization. The developer writes "*When I parallelise with Julia I get a performance degradation, i.e. one process is faster then two processes! I am obviously doing something wrong... I have consulted other questions asked in the forum but I could still not piece together an answer*" The root cause of decreased performance was related to the incorrect usage of Julia code elements: instead of using `SharedArray` or `DistributedArray`, the developer used a self-designed algorithm and data structure.

5. **Type (278)** This category consists of questions related to types in Julia. While Julia supports as many as 221 types [20], when it comes to using these types developers face challenges expressed as SO questions. We observe questions related to immutability and instantiation to be asked on SO. For example, in a SO question ($_{31775391}$), a developer seeks to learn about the performance implications of using immutable types in Julia. According to the SO discussion, immutable types are "*fast when they are small and consist entirely of immediate data, with no references (pointers) to heap-allocated objects*". As another example, in a SO question ($_{29261431}$) a developer wanted to know how to provide keyword arguments when instantiating a self-defined type.

6. **Regular Expression (245)** This category consists of questions that are related to using regular expressions. From our analysis, we observe developers to ask about Julia-related libraries that can

convert strings into regular expressions ($_{31000633}$), as well as ask about how to perform fuzzy regex matching ($_{37933471}$).
7. **Date Operation (182)** This category consists of questions that are related to performing date-related operations, such as modifying, formatting, and converting dates. For example, in a SO question ($_{27084893}$), a developer asked about how to format date using a certain format. The solution was to use the `Date` constructor.
8. **Cross Language Transfer (133)**: This category consists of questions related to cross-language transfer, i.e, the phenomenon of developers transitioning from one programming language to another [24], e.g., from R to Julia. Cross-language transfer imposes challenges for developer that could lead to undesirable consequences. For example, in a SO question we observe to seek the equivalent code construct for `sapply` that is available in R's base library ($_{30281326}$). In response, another SO user suggested a variety of solutions: (i) use of anonymous functions, (ii) transposing, and (iii) splatting. The discussion in the SO question shows that functions, which are available in one language may not be available in another language, necessitating developers to allocate extra efforts.
9. **I/O Operations (128)** This category consists of questions related to performing input and output operations, such as file reading/writing and directory management. For example, in a SO question ($_{49533361}$) a developer asks about the best practices on how to setup multiple I/O buffers in Julia. In response, a SO user mentioned that use of array comprehensions or the use of `map()` could be helpful.
10. **Web Mining (80)** This category consists of questions that are related to mining content from the web. For example, in a SO question ($_{59010720}$), a developer asks about the return type for `HTTP.request()`, and asked what is the correct content-encoding header to get necessary data.
11. **Domain Specific Language (22)** This category consists of questions related to developing domain-specific languages (DSLs). One such DSL is JuMP, a DSL for mathematical optimization. For example, a SO user, who self-describes as a newcomer to Julia, asks about how to formulate an optimization problem in JuMP ($_{31812458}$).
12. **Metaprogramming (15)** This category consists of questions related to metaprogramming in Julia. Metaprogramming is the technique of where one computer program has the ability to use, read, modify other computer programs and even the program itself [25]. Julia supports metaprogramming using utilities, such as `macros` and `symbols`. However, while using these utilities developers face challenges and ask questions on SO. As an example, in a SO question ($_{30905546}$) a developer incorrectly used symbols, which resulted in returning of incorrect values from a function. `Symbols` in Julia are defined as features used to represent Julia's own code, i.e., represent code constructs, such as assignments, function calls, literals, and variables [3].
13. **Garbage Collection (11)** This category consists of questions related to Julia's internal garbage collector, `GC`. Using `GC`, a developer can allocate and deallocate memory in a Julia program. However, the process of garbage collection in Julia can be confusing to developers as demonstrated in a SO question ($_{47449177}$). The SO user was unaware of the fact that Julia's garbage collector is free to collect a variable at any time after it is last used [3].

4 DISCUSSION

We discuss the findings of our paper as follows:
Implications for toolsmiths We outline the following areas that toolsmiths can focus on:

- **Enhancing Support for Cross Language Transfer**: Shrestha et al. [24] documented evidence related to cross language transfer by analyzing SO questions. We too have documented evidence related cross language transfer that further substantiates findings reported by Shrestha et al. [24]. Existence of cross language transfer showcases that developers can transition from a non-Julia programming language to Julia, but in the process face challenges. Our conjecture is that as more developers transition from established languages, such as from Python to Julia, they will seek information on how they can accomplish tasks in Julia that they were previously able to do before with the language they are transitioning from. Based on our findings, we recommend further systematic analysis of challenges that can occur due to cross language transfer. We also advocate for development and dissemination of documentation-related resources that include examples of other programming languages so that developers can adequately map an example Julia program to a program in a language that they already know.
- **Type Assistance**: Julia supports as many as 221 types [20], which allows developers the ability to accomplish a wide range of computational tasks. However, developers face challenges while using these types as expressed in forms of SO questions. To facilitate developers in writing Julia programs we advocate for development of tools that can nudge developers while performing type-related operations. These tools can provide information on (i) how to correctly use Julia's types, and (ii) automatically fix type-related errors as developers write code.
- **Package Assistance**: Similar to other programming languages, such as Go [28] and Python [19], package-related challenges also accompany Julia development. From our analysis, we observe developers to seek help on using a package correctly, and also using the correct package. Based on our findings, we conjecture that the Julia ecosystem will require derivation of Julia-specific techniques for adequate package resolution.

Similarities with Other Stack Overflow Topics: We observe our identified categories to appear for other technologies as well. For example, date-related operations and package resolution were also identified as question categories for Python [27]. As another example, array-related questions have been reported for big data-related topics on SO [8]. Our findings show questions that are commonplace in data analytics-related fields, such as big data and Python also appear for Julia. As Julia advertises itself as a language that facilitates rapid execution of programs for computationally-heavy tasks, we advocate for systematic mitigation of the challenges that developers face while writing Julia programs.

Despite above-mentioned similarities, certain categories, such as type, metaprogramming, and garbage collection are unique to Julia's design and syntax. Solutions to these question categories require understanding of Julia's syntax, which differentiate them from other programming languages.

5 THREATS TO VALIDITY

We discuss the limitations of our paper as follows:

Conclusion Validity: Our findings are limited to rater bias, as all categories were derived by the first and the last author. The rater may have missed categories due to their subjective bias. We mitigate this limitation using two raters. We also may have missed question categories that might be available in other types of software repositories.

External Validity: We only used SO questions to determine questions categories. Our analysis is susceptible to external validity as other categories might be available through analysis of other question and answer websites.

Internal Validity: Our derived categories, and the mapping of SO questions to identify categories are susceptible to internal validity because the rater may have inherent expectations on the outcome of the process.

6 CONCLUSION

Julia is an emerging programming language, which is perceived to solve the 'two language problem'. Despite reported benefits, such as rapid program execution and productivity, developers face challenges in using Julia. We conduct an empirical study with 3,093 SO questions, from which we identify 13 question categories. Our analysis shows developers to encounter a wide range of challenges related to Julia programs, such as visualization, array manipulation, and package resolution. We observe visualization-related questions to be the most frequently occurring category. Based on our findings, we recommend enhancing support for developers with Julia-based tools and techniques for cross language transfer, type-related assistance, and package resolution.

ACKNOWLEDGMENTS

We thank the PASER group at Tennessee Tech University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) through Award # 1852126, # 2043324, and # 2026869.

REFERENCES

[1] [n.d.]. Julia. https://juliacomputing.com/case-studies/celeste.html.
[2] [n.d.]. Julia: Come For The Syntax, Stay For The Speed. https://www.nature.com/articles/d41586-019-02310-3.
[3] [n.d.]. The Julia Language. https://docs.julialang.org/en/v1/.
[4] [n.d.]. Julia Update: Adoption Keeps Climbing; Is It a Python Challenger? https://www.hpcwire.com/2021/01/13/julia-update-adoption-keeps-climbing-is-it-a-python-challenger/.
[5] [n.d.]. Programming Languages: Developers Reveal What They Love and Loathe, and What Pays Best. https://www.zdnet.com/article/programming-languages-developers-reveal-what-they-love-and-loathe/.
[6] [n.d.]. Why Julia is Slowly Replacing Python in Machine Learning and Data Science. https://www.section.io/engineering-education/why-julia-is-slowly-replacing-python-for-machine-learning-and-data-science/.
[7] Vard Antinyan, Miroslaw Staron, and Anna Sandberg. 2017. Evaluating Code Complexity Triggers, Use of Complexity Measures and the Influence of Code Complexity on Maintenance Time. *Empirical Software Engineering* 22, 6 (2017), 3057–3087.
[8] Mehdi Bagherzadeh and Raffi Khatchadourian. 2019. Going Big: A Large-scale Study on What Big Data Developers Ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 432–442.
[9] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. 2018. SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/3196398.3196430
[10] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 120 (Oct. 2018), 23 pages. https://doi.org/10.1145/3276490
[11] Roland Churavy. 2019. *Transparent Distributed Programming in Julia*. Ph.D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/122755
[12] Benjamin Crabtree and William Miller. 1999. *Doing Qualitative Research*. SAGE Publications.
[13] Ioana Dogaru and Radu Dogaru. 2015. Using Python and Julia for Efficient Implementation of Natural Computing and Complexity Related Algorithms. In *2015 20th International Conference on Control Systems and Computer Science*. IEEE, 599–604. https://ieeexplore.ieee.org/abstract/document/7168488
[14] Effat Farhana, Nasif Imtiaz, and Akond Rahman. 2019. Synthesizing Program Execution Time Discrepancies in Julia Used for Scientific Software. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 496–500. https://ieeexplore.ieee.org/abstract/document/8918949
[15] John Gibson. 2017. The Julia Programming Language: The Future of Scientific Computing. *APS* (2017), L39–011. https://ui.adsabs.harvard.edu/abs/2017APS..DFDL39011G/abstract
[16] Jan Gmys, Tiago Carneiro, Nouredine Melab, El-Ghazali Talbi, and Daniel Tuyttens. 2020. A Comparative Study of High-productivity High-performance Programming Languages for Parallel Metaheuristics. *Swarm and Evolutionary Computation* (2020), 100720. https://www.sciencedirect.com/science/article/abs/pii/S2210650220303734
[17] Tomasz Januszek and Mark Pleszczyński. 2018. Comparative Analysis of the Efficiency of Julia Language Against the Other Classic Programming Languages. *Silesian Journal of Pure and Applied Mathematics* 8 (2018). https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-c4339453-4519-4b92-a673-307638a50cb1
[18] Klaus Krippendorff. 2018. *Content Analysis: An Introduction to its Methodology*. Sage publications.
[19] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-Gonzalez. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 439–451. https://doi.org/10.1145/3460319.3464797
[20] Simon Poulding and Robert Feldt. 2017. Automated Random Testing in Multiple Dispatch Languages. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 333–344. https://doi.org/10.1109/ICST.2017.37
[21] Akond Rahman, Effat Farhana, and Nasif Imtiaz. 2019. Snakes in Paradise?: Insecure Python-Related Coding Practices in Stack Overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 200–204. https://doi.org/10.1109/MSR.2019.00040
[22] Johnny Saldaña. 2015. *The Coding Manual for Qualitative Researchers*. Sage.
[23] R. Sells. 2020. Julia Programming Language Benchmark Using a Flight Simulation. In *2020 IEEE Aerospace Conference*. 1–8. https://doi.org/10.1109/AERO47225.2020.9172277
[24] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE*.
[25] Diomidis Spinellis. 2008. Rational Metaprogramming. *IEEE software* 25, 1 (2008), 78–79.
[26] Stack Overflow. 2019. Memory Allocation with Broadcasted Operation. https://stackoverflow.com/questions/57295291/.
[27] Hamed Tahmooresi, Abbas Heydarnoori, and Alireza Aghamohammadi. 2020. An Analysis of Python's Topics, Trends, and Technologies Through Mining Stack Overflow Discussions. *arXiv preprint arXiv:2004.06280* (2020).
[28] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. HERO: On the Chaos When PATH Meets Modules. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 99–111.
[29] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. 2018. Julia Subtyping: A Rational Reconstruction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 113 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276483