DOI:10.1145/3511062

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3511061

Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?

By Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin

Abstract

Once a programmer knows one language, they can leverage concepts and knowledge already learned, and easily pick up another programming language. But is that always the case? To understand if programmers have difficulty learning additional programming languages, we conducted an empirical study of Stack Overflow questions across 18 different programming languages. We hypothesized that previous knowledge could potentially interfere with learning a new programming language. From our inspection of 450 Stack Overflow questions, we found 276 instances of interference that occurred due to faulty assumptions originating from knowledge about a different language. To understand why these difficulties occurred, we conducted semistructured interviews with 16 professional programmers. The interviews revealed that programmers make failed attempts to relate a new programming language with what they already know. Our findings inform design implications for technical authors, toolsmiths, and language designers, such as designing documentation and automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem.

1. INTRODUCTION

Peter Norvig wrote a guide, *Python for Lisp Programmers*,[13] to teach Python from the perspective of Lisp. We interviewed Peter regarding this transition and he described a few challenging aspects of switching to Python such as how lists are not treated as a linked list and solutions where he previously used macros required rethinking. When asked about the general problem of switching programming languages, he said, "Most research is on beginners learning languages. For experts, it's quite different and we don't know that process. We just sort of assume if you're an expert you don't need any help. But I think that's not true!" Peter believes that learning new languages is difficult—even for experts—despite their previous experience working with languages. Is Peter right?

Numerous stories on language transitions suggest that even experienced programmers experience difficulties. For example, a Java programmer who transitioned to Kotlin[24] reports that differences such as reversed type notation and how classes in Kotlin are final by default, made the transition less smooth than expected: "if you think that you can learn Kotlin quickly because you already know Java—you are wrong. Kotlin would throw you in the deep end." Similarly, a programmer experienced in C++ who switched to Rust[4] found that Rust's borrow checker, "forces a programmer to think differently." Transitions across radically different languages are especially difficult. For example, a Java programmer switched to Haskell[10] and expressed that "the easy things are often a bit harder to do in Haskell," and another programmer[16] experienced in procedural languages warned that "[lazy evaluation] can be a bit confusing to understand *how* it works in practice especially if you're still thinking like an imperative programmer." Even languages sharing the same runtime can be problematic: "whenever I pick up CoffeeScript, I feel as if most of my understanding of JavaScript suddenly vanishes into thin air."[14] From these stories, one common refrain occurs: previous programming knowledge is sometimes less helpful than expected, and can actively interfere with learning. This seems counterintuitive. Why can previous knowledge actually make learning harder and not easier?

In psychology and neuroscience, interference theory has helped explain the confusion that can occur when older information interacts with newer information.[23] To illustrate, suppose the bread aisle of your favorite store was recently moved. You may reflexively start walking toward the old location due to *interference*—when previous knowledge disrupts recall of newly learned information. However, if you recently saw that the impossible burger was added to the store, using knowledge that frozen food can be found in the frozen section—and not a separate health aisle—is an example of *facilitation,* where your previous knowledge helps retrieval of new information. In the same vein, when a Java programmer is learning Kotlin, we postulate that their prior Java knowledge either facilitates or interferes with learning. The knowledge that Java is objected oriented and uses static typing facilitates their learning as Kotlin shares similar properties. The knowledge that Java classes are not `final` by default interferes with their learning because Kotlin classes are `final` by default.

If previous programming knowledge can be framed as a source of interference with new programming language

The original version of this paper was published in *Proceedings of the the 42nd ACM/IEEE International Conference on Software Engineering*, 2020, 691–701.

acquisition, interference theory can explain why programming language learning can be difficult for experienced programmers. And when previous programming knowledge is not relevant, learning can also be difficult because this knowledge does not facilitate.

To investigate our hypothesis, we first looked for evidence that programmers could have difficulty learning another language due to interference from their previous knowledge. To this end, we conducted an empirical study examining questions posted on a popular question-and-answer site, Stack Overflow.[a] We analyzed 450 posts for 18 different programming languages and qualitatively coded each post, characterizing posts in terms of whether or not programmers made incorrect assumptions based on their previous programming knowledge. Then, to understand what learning strategies programmers used when learning another language—and why previous knowledge could interfere with this process—we interviewed 16 professional programmers who had recently switched to a new programming language.

We found that:

- Cross-language interference is a problem: 276 (61%) cross-language posts on Stack Overflow contained incorrect assumptions due to interference with previous language knowledge.
- Based on our interviews, professional programmers primarily learned new languages on their own, using an opportunistic strategy that often involved relating the new language to previous language knowledge; however, this results in interference which harms their learning.
- Learning a new language involves breaking down old habits, shifting one's mindset, dealing with little-to-no mapping to previous languages, searching for proper documentation, and retooling in a new environment. All together, these challenges make learning another language difficult.

2. METHODOLOGY

To explore how programmers learn a new language, and understand their potential sources of confusion, we conducted a mixed-method study through an empirical investigation of Stack Overflow posts across various languages and through semistructured interviews. We do so through the following research questions:

2.1. Research questions

- **RQ1: Does cross-language interference occur**? We examined questions programmers had about programming languages on Stack Overflow for evidence of interference with previous programming knowledge.
- **RQ2: How do experienced programmers learn new languages?** To gain a better understanding of why cross-language interference occurs, we interviewed professional programmers on how they learn new languages.
- **RQ3: What do experienced programmers find confusing in new languages?** To examine the ways in which programmers mix a new language with their previous knowledge, we asked programmers about obstacles they faced, and surprises they encountered in their new languages.

a https://www.stackoverflow.com

2.2. Phase I: Study design for Stack Overflow

To answer RQ1, we conducted a study using Stack Overflow posts.

Data collection. To gather Stack Overflow questions, we used the SOTorrent[2] data source from the 2019 MSR Mining Challenge. We queried 26 programming languages used previously by Erik[b] and Waren[c] in their investigation of popular language migrations, based on Google search keywords and GitHub repositories. We gathered Stack Overflow questions for each `<language A, language B>` pair. To keep the analysis tractable, we considered only the association between the two languages, and not the direction of the possible interference. We used a stop-rule criteria to cover over 95% of total posts, which resulted in 15 out of the 26 language pairs as shown in Table 1. The materials for the study are available online.[d]

Query criteria. We used BigQuery[e] to query the SOTorrent database and used the following filtering criteria to capture potential posts where the programmers are asking questions about a new language (target) coming from a previous language (source):

1. The question is tagged with both languages, or
2. The question is tagged with the source language but contains the text of the target language in the title or body, vice versa .

Analysis. To understand whether or not cross-language

b https://erikbern.com/2017/03/15/the-eigenvector-of-why-we-moved-from-language-x-to-language-y. html
c https://blog.sourced.tech/post/language_migrations/
d https://go.ncsu.edu/cross-lang-study
e https://cloud.google.com/bigquery/

Table 1. Posts by language pair.

Language pair[1]	Posts[2]	% Accepted[3]	*n*	Correct[4] %
<C, C++>	30863	65%	9	30%
<C#, Visual Basic>	11522	62%	8	27%
<Objective-C, Swift>	9416	50%	10	33%
<Python, C++>	6763	51%	15	50%
<Java, C#>	6748	59%	16	53%
<Scala, Java>	6622	55%	8	27%
<PHP, Java>	6152	46%	16	53%
<R, Python>	2824	49%	12	40%
<Kotlin, Java>	2565	53%	6	20%
<Matlab, Python>	2407	53%	11	37%
<Node, PHP>	2077	40%	14	47%
<Ruby, Python>	1314	65%	14	47%
<Perl, Python>	1152	67%	13	43%
<Lua, C++>	1143	63%	12	40%
<Clojure, Java>	1098	68%	10	33%

[1] The pair of programming languages.
[2] Total number of questions where the two languages are tagged or referenced in body.
[3] Percentage of questions that have accepted answers.
[4] Total posts (out of 30) classified as having correct assumptions formed from prior language knowledge.

interference occurs, we performed a manual inspection of Stack Overflow posts (see Table 1). We inspected a random sample of 30 posts for each pair to keep categorization tractable, as done in the works of Barik et al.[3] We manually excluded posts that did not make any explicit connection between the languages of each pair, sampling another random post to replace it as necessary. Because the inclusion and exclusion criteria can have multiple interpretations, the first two coauthors labeled a random sample of 30 posts. This labeling had 100% agreement between the coauthors, and suggests a clear understanding of how to categorize posts. The two coauthors proceeded to label the rest of the Stack Overflow posts using two classifications:

- *Correct:* The post makes a connection to a previous programming language with correct assumptions regarding the target language as revealed by the accepted answer, or
- *Incorrect:* The post makes a connection to a previous programming language with incorrect assumptions regarding the target language as revealed by the accepted answer.

Next, we calculated and reached agreement on the interrater reliability (IRR) between the two coauthors (Cohen's $k = 0.89$). Disagreements on the classification—even after discussion—was reconciled by the first author. Finally, we calculated the percentage of correct and incorrect posts, using instances of correct and incorrect assumptions as evidence of cross-language interference and facilitation.

2.3. Phase II: Study design for interviews with professional programmers

To answer RQ2 and RQ3, we conducted semistructured interviews with professional programmers.

Participants. We used *purposive sampling*[21] to recruit 16 professional programmers who were learning a new programming language within the past six months (see Table 2); these participants were still early in their learning process and working through their initial stumbling blocks in the new language. The participants (12 male, 4 female, self-reported) were from large software, technology, and data analytics companies with years of programming experience ranging from 5 to 31 years ($\mu = 12.8$, $sd = 6.6$). There were a total of 14 unique language transitions. Before the interview, participants completed a background questionnaire asking them about their previous languages and an obstacle they have experienced although adapting to the new language.

Protocol. We conducted semistructured interviews either onsite or remotely, within 60 min time blocks. Two of the authors conducted and recorded the interviews separately. All sessions were conducted with a single observer and a single programmer on the following topics: (1) participant background, (2) first steps, (3) obstacles, (4) learning process, and (5) general strategies. The background information from the questionnaire was used to tailor the questions for the participants. The semistructured interview format allowed the flexibility to ask questions impromptu and dig deeper into more specific obstacles. The recordings were later transcribed by the first author for analysis.

Analysis. *RQ2: How do experienced programmers learn new languages?* To answer RQ2, we conducted inductive thematic analysis[9] on the interview transcripts over multiple phases: transcribing interviews, generating open codes by labelling notable recurring statements made by the participants, identifying relationships between the codes, and organizing them into meaningful themes.

RQ3: What do experienced programmers find confusing in new languages? To understand how programmers confuse language concepts, we selected themes from our analysis that highlighted interference due to previous programming knowledge.

Table 2. Participants.

ID	Exp[1]	Domain	Recent transition
P1	15	Compilers	C# ⇒ Python ⇒ C++
P2	9	Data Science	Python ⇒Julia
P3	18	Information Sciences	Python ⇒ PHP
P4	15	Neuroscience	R ⇒ Python
P5	10	Security	C++ ⇒ TypeScript
P6	20	Cloud Services	C# ⇒ TypeScript
P7	6	Cloud Services	C# ⇒ Python
P8	10	Web Platform	C# ⇒ JavaScript
P9	31	Data Science	C# ⇒ JavaScript ⇒ Scala
P10	8	Business Applications	C# ⇒ Rust
P11	12	Web Platform	C# ⇒ Ruby
P12	10	Data Science	Python ⇒ SAS
P13	6	Software Engineering	C++ ⇒ JavaScript
P14	10	Data Science	R ⇒ Python
P15	20	Software Engineering	C# ⇒ Swift
P16	5	Data Science	R ⇒ Python

[1] Years of self-reported programming experience.

3. RESULTS

3.1. RQ1: Does cross-language interference occur?

Cross-language interference occurs on Stack Overflow across various language pairs. We found a total of 276 instances of incorrect assumptions (see Table 1), which is around 61% of the 450 posts inspected. There were a total of 174 posts with correctly stated assumptions, which is only around 39% of the total posts. It is important to note that this provides evidence of interference occurring but does not imply programmers have incorrect assumptions 61% of the time. The `<Kotlin, Java>` pair had the highest number of posts with incorrect assumptions, which reflects the Java programmer's confusion mentioned in Section 1. The next two pairs, `<C#, Visual Basic>` and `<Scala, Java>`, also contained a high number of incorrect assumptions. However, there were other pairs such as `<Python, C++>`, `<Java, C#>`, and `<PHP, Java>`, which had a more even distribution of posts with correct and incorrect assumptions; this suggests easier transitions between the languages. Although reviewing the 450 Stack Overflow posts, we encountered instances where programming languages behaved in surprising ways for programmers. We highlight three examples, two of which involved interference between syntax and concepts, and one which involved facilitation—making it easier to use type inference.

Interference: R ⇒ Python[f]

An R programmer is now using Python and its data processing library, Pandas. They are unable to successfully relate their previous knowledge about subsetting, in R, to Python: "I'm seriously confused. Maybe I'm thinking too much in R terms and can't wrap my head around what's going on in Python."

They present the R expression they want to translate, as well as several attempted translations in Python:

```
# R
data[data$x > value, y] <- 1
# Python
data['y'] [data['x'] > value] = 1
```

Several concepts in R interfered, but we will highlight the most significant: Python prevents assignment to copies of dataframes. In this case, the indexing operation `data['y']` returns a copy of the dataframe and setting the value with `[data['y'] > value] = 1` will not work as the R programmer expects. The knowledge that the equivalent R expression will set the value of 1 without any warnings interferes with Python's warning.

Interference: PHP ⇒ JavaScript[g]

A PHP programmer who has switched to programming in JavaScript asks how to store transient information (sessions), such as application state about a user. Typically, PHP uses server-side session variables (`$_SESSION`) for this purpose. Although related concepts, such as local storage and browser-based sessions exist, the programmer is warned that sessions cannot be safely and securely stored directly on the client—the programmer's knowledge about server-side sessions leads to a faulty assumption about their applicability in other programming contexts.

Facilitation: Java ⇒ Kotlin[h]

A Java developer is learning Kotlin. They ask if the following Kotlin expression can be simplified:

```
val boundsBuilder: LatLngBounds.Builder =
  LatLngBounds.Builder()
```

The developer suspects their declaration is more verbose than it should be, given their knowledge of local variable type inference in Java. They assume the declaration can be simplified:

```
val boundsBuilder = LatLngBounds.Builder()
```

This is an example of facilitation—the accepted answer confirms that the developer can simplify the expression because Kotlin supports type inference, allowing for the explicit type declaration to be removed.

These examples illustrate how previous knowledge of language syntax and concepts interact with knowledge learned in a new language. In some cases, this results in interference, which harms a programmer's ability to grasp new syntax and concepts in the new language. In other cases, this results in facilitation, which helps programmers make meaningful connections to previous languages and helps them learn the new language.

Cross-language interference occurs across various language transitions on Stack Overflow posts. We found that 61% of the 450 posts contained incorrect assumptions about the target language, and only 39% contained correct assumptions.

3.2. RQ2: How do experienced programmers learn new languages?

In this section, we present the themes on how experienced programmers learn new languages.

Programmers learned languages on their own. Programmers who switched teams lacked formal training for the new language and its associated technology stack, leaving learning to themselves. For example, when P1 switched from C# to Python for a new project, there was not any training involved and the onboarding process was, "hey we want to get exposed to the Python world, go get started!" Although some programmers were given training initially on the project, "realistically for learning the new language [they] were pretty much on [their] own" (P7). This forced programmers to watch "language tutorial videos on Plural-sight"[i] (P5) or read online documentation. Some programmers "got initial tips from some folks from the team on what's what" (P6), and when running into complex issues "reached out to the group and said has somebody else hit this before?" (P1).

Just-in-time learning is a dominant strategy. To learn new languages, every programmer we interviewed used *just-in-time learning*,[8] an opportunistic strategy focused on only learning features as needed. Given time constraints, programmers made use of immediately available resources such as online documentation, video tutorials, online searches, and available experts. Traditional resources such as programming language books were only used as a reference, because programmers "just don't have time to do that" (P5). Programmers were primarily concerned with completing tasks in a reasonable time and "figuring out how to not burn tons of time on a single problem" (P1). Quicker resources, such as cheat sheets, were preferred for language transitions. For example, the first thing P2 did was to make use of cheat sheets to help them transition from Python to Julia. P15 was also a fan of cheat sheets:

> *It seems like if you were going from one framework to another, from one technology stack to another—even if you 're not going from A to B, you 're just starting off on B—there's probably a content cheat sheet that every dev needs to know.* (P15)

f https://stackoverflow.com/questions/30923882
g https://stackoverflow.com/questions/47137666
h https://stackoverflow.com/questions/38131655

i https://www.pluralsight.com

Programmers related, the new language to previous languages. To help accelerate the learning process, programmers generally tried to relate the new language to their previous languages. Programmers started by "loosely taking ideas from working in another language" (P14) or looking at existing code because "it's already probably been written and it's out there somewhere or at least something close to it" (P1). Although this learning strategy was useful for bootstrapping, some programmers started from scratch. For example, when moving from C# to Ruby, P11 described "trying to be very conservative and mindful and trying not to map anything over, but just treating everything as something brand new." Similarly, P12 explained that they did not try to map things from Python when learning SAS "mostly because the syntax was so new that every time [they] tried to do anything, [they] would have to go and google the syntax." P10 expressed a similar problem when learning about managing memory in Rust after years of using C#: "there wasn't a clean way for me to just get there. I had to go and learn that stuff from scratch." These examples illustrate that programmers typically try to reuse knowledge (facilitation), but sometimes avoid doing so when it's more troublesome.

Programmers use an opportunistic learning strategy, relating syntax and concepts of the new language with their previous language. This offers expediency but causes interference when major differences exist between the two languages.

3.3. RQ3: What do experienced programmers find confusing in new languages?

In this section, we present the themes explaining how programmers confuse language concepts.

Old habits die hard. Programmers had to constantly suppress old habits acquired from previous languages. For example, P3—who was used to Python—had trouble adapting to block delimiters in PHP, where "it's near-impossible to figure out exactly which opening brace you're closing once your HTML/PHP gets to any complexity at all." Similarly, P15 realized that "in Swift, the open curly bracket needs to be on the initial line of the method declaration and if you put it on the next line the method may not execute in an expected fashion." Differences such as 0 versus 1 indexing for lists between languages such as Python and R caused frustrations for P4: "typing `a[1]` thinking that it's `a[0]`, and then wasting 5 min such as a complete fool not understanding why nothing makes sense." Programmers are able to resolve these small differences, but it still causes interference at the onset of learning a new language.

Mindshifts are required when switching paradigms. Some language transitions required fundamental shifts in mindsets, or "mindshifts."[1] For example, when P2 transitioned from Python to Julia, they were constantly trying to make an object and realizing that "there's no objects, there's only structs!" With Julia, they needed to write more functional code, a shift from the object-oriented programming that they were used to in Python: "it was just needing to shift that and realize I'm never gonna write 'something-dot-something-else' ever or rarely." P10 had to completely rethink the problems they solved in C# when switching to Rust due to the ownership feature for memory safety:

> *A really fascinating thing about learning Rust was that when I went and started to do these things—things that I would reach for in C# that I knew would work—Rust wouldn't allow it and as a result I had to rethink the problem and re-implement it in a way where the ownership characteristics of that algorithm were very explicit.* (P10)

Large paradigm shifts occurred for P5, P6, and P13—all transitioning from imperative or object-oriented coding to event-driven and asynchronous coding—forcing them to think differently. The programmers had to learn brand new concepts in JavaScript such as asynchronous programming or "shadow and virtual DOMs" (P13). P6 described how it was difficult making sense of asynchronous code because "you got a whole bunch of '`async/await` mode' working in your mind and you have to convert it." To make matters worse, "the most confusing part is there are a couple of ways to do asynchronous programming, with observables or promises" (P13). For P5, the front-end coding in TypeScript was a big challenge because "for the back-end, the code I think is more straightforward. You have the logic and most likely you know single places you'll handle it. It's not like the UI." Here, the interference issues aren't due to any particular syntax or concept but the way one solves problems in the new language.

Learning a language is difficult when there is little to no mapping with previous languages. Programmers had a harder time learning the new language when there was little to no mapping of features to previous languages. For example, P12 could not make sense of some fundamental programming language features of SAS that were clear in Python, such as statements versus method parameters. They could not understand "why some things are statements that affect a procedure, but aren't parameters" and were "still confused about the overall syntax and what is or isn't a statement"—even after having worked in the language for a few weeks. A drastic example was P5, who experienced a big transition from C++ to TypeScript, resulting in *tech shock*: "Everything is different! Not just the programming language—the IDE, source control, everything is different." Similarly, P13 found that concepts were challenging in JavaScript because they "could not equate it back to C++."

In the extreme case, programmers were forced to learn completely foreign syntax or concept. For example, P9 had difficulty learning traits in Scala because they "never had a language with traits before. Traits have a default implementation and understanding what would be performant and what wouldn't—and when to use what—that was the tricky part." P7 learned that for Python, "the major difference is the multiple inheritance thing, that Python inherits from the C++ world, which supports multiple inheritance. In C# you can't do that." In another case, the difficulty was due to differences in memory management, for example, when P10—who previously used C#—was learning Rust:

> *There's a very alien concept in Rust that is the borrow checker, which is the concept of having the compiler verify more things,*

and the way it does it is somewhat esoteric. That's very alien, and that's something that I think is really cool but it's also very rough at the moment and so that's kind of something that's been the biggest struggle when trying to learn Rust. (P10)

The lack of mapping caused a lot of confusion even within the same context. For example, P14, who switched from one data analysis language (R) to another (Python/Pandas), could not find an immediate equivalent for R's `spread` and `gather` functions: "Pandas already had the functionality but it was more hidden using drop level and unstack. These were really hard to understand in Pandas—it was some pretty weird stuff." Similarly, P15, who switched from C# to Swift, was very surprised to learn how the user interface code and its graphical layout view in Xcode were connected: "Knowing that you can't interact with a UI object straight out of the box from the code is very important. Once you draw the referencing outlet connection between View and Controller you can trigger methods and `get`/`set` properties as you'd expect in the .NET world."

Searching for the right terminology and code examples is difficult. We found that moving to a new programming language made it difficult to search for information about the language and its associated technologies. Programmers had trouble acquiring the vocabulary even before performing the search. For P12, the names for the same structures in Python were slightly different than SAS where a "dataframe is data set, a row is an observation, a column is a variable." Searching was difficult because on the one hand, "it's the breadth of the libraries that usually get you, you don't even know what exists, what to even look for to see if something is already there" (P1). On the other hand, insufficient search results provided little to no facilitation. P4 had difficulty searching information for the Python library `seaborn`—compared to the equivalent R library `ggplot`—because "for ggplot, if you google anything, you get like 100 hits, and the top ones are bound to be good due to Google selection of results. With seaborn, you get like 10 hits."

Even when programmers found documentation and code examples, they were either incomplete or lacking in detail. P2 was frustrated with the Julia documentation because "it was so useless for figuring out the imports." Similarly, P12 expressed that the SAS documentation "only tells you how to copy-paste and run a simple program, leaving you completely mystified as to how the execution and control flow of a SAS program works." This lead to frustration, especially when better documentation existed in previous languages: "Xcode documentation samples were pretty good enough to where they would run. But the documentation, MSDN, and the available samples for creating Microsoft platform-based applications were tenfold deeper and richer and easier for to use." (P15)

Retooling is a necessary and challenging first step. Finally, before programming in the new language, programmers faced difficulty retooling themselves in a new environment. This typically involved adapting to the discrepancies of the new integrated development environment (IDE) for programming in the language. Although programmers were able to adapt to basic features of IDEs (facilitation), there was interference when some aspects of the IDE differed from their previous IDEs. For example, P15 discovered that in Xcode "build targets aren't 'Universal' in definition (like .NET) and when terminologies are shared across platforms but don't implement the same notion, you're lost for days!" Interestingly, for P9 there was interference when they tried building their Scala project in IntelliJ because the IDE attempted to support Scala, but continued presenting dialogs in the previous language:

Part of the problem is IntelliJ is aimed at the Java developer and I'm using SBT, which is from the Scala world. And it's sort of importing the SBT into the concepts in the IDE of IntelliJ. So I'm looking at dialogs that are all about Java and which JDK and that doesn't map to what I wrote in the declarative SBT language. (P9)

Other concerns regarded either a lack of IDE features or learning new features that were distracting. P2 had been "spoiled with Python and PyCharm" and found it very difficult to find proper IDE support for Julia; they just wanted "an IDE that does syntax highlighting and IntelliSense-like autocompletion." P1 found that learning a new feature—such as debuggers—are counterproductive "because you're learning and debugging at the same time as opposed to just debugging once you're fluent." However, sometimes the transition to new tools in the language also benefited programmers:

I think right now the build system for us, I think it's better since now we are using DevOps—a pipeline to build the code. It's very easy for us to even schedule the private build and also it's very easy for us to quickly get new things, check in the code, test it, and even build things on top of it. (P5)

Programmers confuse a new language's syntax and concepts with previous languages, leading to a number of issues such as trying to suppress old habits, wrestling with mapping issues, struggling to find and use proper documentation, retooling and shifting one's mindset for new paradigms.

4. LIMITATIONS

Our mixed-methods approach of investigating Stack Overflow and conducting interviews introduces certain trade-offs and limitations.

Our sampling approach for Stack Overflow targets diversity (rather than representativeness) in order to identify evidence of interference across many different programming languages. The posts we examined on Stack Overflow as well as our interviews do not completely cover the set of all language transitions, as the full permutation space of language transitions is intractable. Our approach attempts to cover transitions that are most likely to occur in practice, which means we might miss other interference issues.

We used correct and incorrect assumptions as a proxy construct for facilitation and interference. Although this approach provided a useful, high-level characterization of the Stack Overflow posts, open coding—a more intricate qualitative coding technique—might have provided further insights. However, open coding is significantly more costly to execute, and we conducted semistructured interviews with experienced programmers to delve deeper into cross-language interference.

Finally, qualitative research involves not only the qualitative data under investigation but also a level of subjectivity and interpretation on the part of the researcher for framing and synthesizing the results. To support interpretive validity, we performed a single-event member check with our results: six participants who replied agreed with our presentation of the results and only wanted minor changes to their quotations. Other theories such as *notional machines,* could have also been used to identify and explain confusions when learning a programming language.[6]

5. RELATED WORK

Novice misconceptions. Programmers often have misconceptions although learning new programming languages, but most studies have focused on novices. Swidan et al.[20] propose "intervention methods to counter those misconceptions as early as possible," but this work is primarily targeted to novices. Similarly, Kaczmarczyk et al.[12] have examined misconceptions and how to measure them for novices. By contrast, the novelty of our work is toward experienced programmers who need to switch languages, and requires methods of learning distinct from those designed for novices. Our study investigated switching languages for experienced programmers and examined how knowledge of previous languages interferes when learning another language.

Programming language transitions. There are a few studies on transitions between programming languages. Scholtz and Wiedenbeck[15] studied experienced Pascal or C programmers writing a program in a new language, Icon, and found that they were strongly influenced by their knowledge of what would be appropriate in previous languages. Similarly, Uesbeck et al.[22] studied the effect of using multiple languages (SQL and Java) in a controlled study, and although the results were inconclusive, the authors suggest that the randomized controlled trial methodology could be effective for studying the productivity costs associated with mixing languages. For interventions, Bower et al.[7] explored a new teaching approach called Continual And Explicit Comparison (CAEC) to teach Java, using facilitation, to students who have knowledge of C++. They found that students benefited from the continual comparison of C++ concepts to Java. Shrestha et al.[17] used a similar technique using a tool called Transfer Tutor to teach R from the perspective of Python; programmers who used the tool found the comparisons between the languages useful. We used the lens of interference theory to uncover interference issues in the modern context, examined numerous language transitions, and found other issues that have not been explored such as little to no mapping of language features (Subsection *Learning a language is difficult when there is little to no mapping with previous languages*) and retooling (Subsection *Retooling is a necessary and challenging first step*).

Programming knowledge. Researchers have suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. *Programming plans* are schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code.[19] For example, in the works of Bellamy and Gilmore,[5] the authors examined the protocols generated from experts in different languages as they created programs, different programming language experts generated different types of representations. We believe our results provide further insight as to why plans may not generalize across languages: we found programmers tend to relate a new language to previous languages (Subsection *Programmers related, the new language to previous languages*) (reusing previous programming plans), but due to interference issues, the previous plans might either need significant modifications (Subsection *Learning a language is difficult when there is little to no mapping with previous languages*) or be replaced entirely (Subsection *Mindshifts are required when switching paradigms*), depending on how closely related the two languages are.

6. DISCUSSION AND IMPLICATIONS

Our findings demonstrate that interference is a widespread phenomenon, forcing programmers to adopt suboptimal, opportunistic learning strategies. In Stack Overflow, instances of interference are found across all of the programming languages we investigated. Furthermore, in our interviews, participants reported that interference arises routinely as they learn a new language—for example, from having to suppress old habits from previous languages (Subsection *Old habits die hard*) or having to "rethink the program" (P10) due to a substantially different paradigm (Subsections *Mindshifts are required when switching paradigms* and *Learning a language is difficult when there is little to no mapping with previous languages*). As opposed to learning "step-by-step" (P5), experienced programmers in our study used opportunistic strategies to learn essentially "on [their] own" (P7) or "learning through work" (P13), for example, using online resources or asking teammates (Subsection *Just-in-time learning is a dominant strategy*). Unfortunately, these informal approaches to learning sometimes result in an incomplete lens for how the language works, resulting in "unintentional bugs" (P5) and other difficult-to-diagnose problems in the code when something does not work as expected.

In the remainder of this section, we present design implications for technical authors, toolsmiths, and programming language designers that can help reduce some of these interference issues for programmers.

Implication I—Design documentation that reduces interference and supports knowledge transfer. Programmers in our study desired more accessible resources that leveraged the programming knowledge they already have (Subsections *Just-in-time learning is a dominant strategy* and *Programmers related, the new language to previous languages*). Such resources included "cheat sheets," which present code snippets that map their familiar language to their new language (P2) and relate concepts they already know "from working in another language" (P14), to the new language tutorials, and even resorting to "reading other people's code" (P3, P15) to understand the programming language idioms. Our findings suggest that resources that teach languages through relating a new language to a known language are more useful and accessible to programmers than resources that present the new programming language in isolation.

However, these resources are handcrafted using the authors' intuitions about potential misconceptions, and not necessarily the ones that programmers actually have. Although there is prior work on misconceptions about novice programmers,[12] misconceptions experienced programmers have are comparatively understudied. Shrestha et al.[18] presented three possible instrument designs that can be used for discovering and validating misconceptions when switching languages for experienced programmers. Such research is needed to make learning resources more effective and relevant to experienced programmers.

Implication II—Build automated tools to provide on-demand feedback. Automated tools can help programmers avoid the context switch involved with reading technical documentation by providing information in their program environment as they work (Subsections *Just-in-time learning is a dominant strategy* and *Searching for the right terminology and code examples is difficult*). For example, Johnson et al.[11] propose "bespoke" notification tools that provide adaptive feedback to the programmer based on the programmer's prior knowledge of programming languages and concepts. Python 3 adopts this idea of using prior programmer knowledge to assist programmers who come from a Python 2 background, through hard-coded error messages:

```
>>> print "Hello"
  File "<stdin>", li
     print "Hello"
                 ^
SyntaxError: Missing parentheses in call to
'print'.
  Did you mean print("Hello")?
```

The SyntaxError message makes the assumption that this error is due to a misconception (print as a statement) instilled from experience with Python 2. We can repurpose this idea generally to language transitions and help programmers more efficiently resolve error messages.

Implication III—Be intentional about programming language syntax, semantics, and pragmatics. Certain programming languages anticipate that new adopters arrive through common pathways. We expect most new Rust users to come from systems programming languages such as C++, and we expect most new TypeScript users to come directly from JavaScript. For these users, intentionally designing language features by considering interference effects can reduce barriers (Subsections *Mindshifts are required when switching paradigms* and *Learning a language is difficult when there is little to no mapping with previous languages*) to adopting the new programming language.

However, designing programming languages for common pathways requires careful design. For example, the borrow checker—a compile-time feature that helps enforce safe memory management[25]—presents a substantial barrier to new Rust users and is "a very alien concept" (P10). The borrow checker seems similar to existing models, such as "resource acquisition is initialization" (RAII), in C++, but ultimately functions differently enough that it can *interfere* with other programmers' past knowledge. As another example, TypeScript is designed to providing static typing to JavaScript and reducing the need to rewrite code for a smoother transition. But providing this smooth transition has a costly consequence: "the TypeScript type system is not statically sound by design."

To improve programming language usability, we need to consider how our language design decisions interfere or facilitate with our anticipated programmers' prior knowledge.

Implication IV—Support not only programming languages, but programming language ecosystems. Issues with interference when learning new programming languages are exasperated when new programming languages bring with them new programming language *ecosystems*—that is, "everything is different, not just the programming language" (P5), but the environment in which the programmer builds, edits, debugs, and tests their code (for example, *tech shock,* Subsection *Retooling is a necessary and challenging first step*).

To address these challenges, we recommend toolsmiths and language designers build tools to bootstrap programmers or unify the tooling environment. For example, React developers provide tool support to welcome programmers into the new ecosystem. Specifically, the `create-react-app`[10j] is an integrated toolchain that abstracts away the complexities of third-party library management, live-editing, optimization, and configuration. `create-react-app` allows the user to quickly and easily begin experimenting with the library until the programmer is comfortable enough to *eject* from the `create-react-app` toolchain. Rather than providing custom tool and editing experiences for IDEs, we recommend solutions such as the language server protocol (LSP),[k] which allows programming language support to be implemented and distributed independently of any given editor or IDE, as long as that IDE implements LSP.

In short, language designers should collaborate with tool designers so that programmers can more easily adopt new programming languages through editing environments that are already familiar to them.

7. CONCLUSION

In this study, we conducted a mixed-method study to understand what impact previous programming language experience has on programmers. We conducted an empirical study of misconceptions found in Stack Overflow questions across 18 different programming languages and semistructured interviews with 16 professional programmers. From Stack Overflow, we found 276 instances of interference that occur across multiple languages. We then interviewed programmers who reported various challenges learning a new language—such as mixing up the syntax and concepts with their previous programming languages—due to interference. We discussed design implications for technical authors, toolsmiths, and language designers, such as designing documentation and building automated tools that reduce interference, anticipating uncommon language transitions during language design, and welcoming programmers not just into a language, but its entire ecosystem. To answer the question posed in the prelude, even professional

j https://create-react-app.dev
k https://langserver.org/

programmers have difficulties with learning programming languages, and we should offer tools and techniques to help them learn more efficiently and effectively.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 1559593, 1755762, and 1814798. C

References

1. Armstrong, D.J., Hardgrave, B.C. Understanding mindshift learning: The transition to object-oriented development. *MIS Q. 31*, 3 (2007), 453–474.
2. Baltes, S., Treude, C., Diehl, S. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. CoRR, abs/1809.02814 (2018).
3. Barik, T., Ford, D., Murphy-Hill, E., Parnin, C. How should compilers explain problems to developers? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), ESEC/FSE, New York, NY, USA, 633–643.
4. Barragan, N. *My Experience with Learning Rust*. 2018.
5. Bellamy, R. and D. Gilmore. Programming plans: internal or external structures. *Lines Thinking: Reflections Psychol. Thought 2*, (1990), 59–72.
6. Berry, M., Kölling, M. The state of play: A notional machine for learning programming. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (2014), Association for Computing Machinery. New York, NY, USA, 21–26.
7. Bower, M., McIver, A. Continual and explicit comparison to promote proactive facilitation during second computer language learning. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11 (2011), Association for Computing Machinery. New York, NY, USA, 218–222.
8. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2009), Association for Computing Machinery. New York, NY, USA, 1589–1598.
9. Braun, V., Clarke, V., Hayfield, N., Terry, G. *Thematic Analysis*. Springer Singapore, Singapore, 2019, 843–860
10. Coelho, L.P. *I Tried Haskell for 5 Years and Here's How It Was*, 2017.
11. Johnson, B., Pandita, R., Murphy-Hill, E., Heckman, S. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, New York, NY, USA, 878–881.
12. Kaczmarczyk, L.C., Petrick, E.R., East, J.P., Herman, G.L. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (2010), ACM, New York, NY, USA, 107–111.
13. Norvig, P. *Python for Lisp Programmers*, 2000.
14. Paskvan, D. *Why Coffeescript?*, 2014.
15. Scholtz, J., Wiedenbeck, S. Learning second and subsequent programming languages: A problem of transfer. *Int. J. Hum.–Comput. Interact. 2*, 1 (1990), 51–72.
16. Shankar, H. Why Learning Functional Programming and Haskell In Particular Can Be Hard, 2011.
17. Shrestha, N., Barik, T., Parnin, C. It's like python but: Towards supporting transfer of programming language knowledge. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2018), IEEE, Manhattan, NY, U.S., 177–185.
18. Shrestha, N., Parnin, C. Instrument designs for validating cross-language behavioral differences. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2019), IEEE, 205–209.
19. Soloway, E., Ehrlich, K., Bonar, J. Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, CHI '82 (1982), 52–57.
20. Swidan, A., Hermans, F., Smit, M. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (2018), ACM, 151–159.
21. Tongco, M.D.C. Purposive sampling as a tool for informant selection. *Ethnobotany Res. Appl. 5*, (2007), 147–158.
22. Uesbeck, P.M., Stefik, A. A randomized controlled trial on the impact of polyglot programming in a database context. In T. Barik, J. Sunshine, S. Chasins, editors, *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*, volume 67 of *OpenAccess Series in Informatics (OASIcs)* (2019), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pages 1:1–1:8.
23. Underwood, B.J. Interference and forgetting. *Psychol. Rev. 64*, 1 (1957), 49.
24. Walacik, B. *From Java to Kotlin and Back Again*, 2018.
25. Zeng, A., Crichton, W. Identifying barriers to adoption for rust through online discourse. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools* (2019), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15.

Nischal Shrestha, Colton Botta, and Chris Parnin ([nshrest, cgbotta, cjparnin]@ncsu.edu), NC State University Raleigh, NC, USA.

Titus Barik ([titus.barik]@microsoft.com), Microsoft Redmond, WA, USA.