Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Investigation of Cultural Differences between Rust and Go Programming Communities

Leon Martin Busch-George
Matrikelnummer: 4663608
lgeorge@inf.fu-berlin.de

Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter/in: Prof. Dr. Claudia Müller-Birn

Berlin, May 2, 2023

Abstract

Background Programmer behavior is not usually thought of as being a cultural phenomenon. There is little structured knowledge on how behavior might relate to social identity in a general software context.
Objectives This research aims to both show there is a meaningful relation between programmer behavior and their communities' culture and to explore methods for learning about them by investigating differences between Go and Rust programmers.
Methods An iterative approach was used both to find differences and how to learn about them. Community research and interviews were alternated to generate and gradually refine hypotheses.
Results Interviews were the most successful for advancing hypotheses. Community and platform research was crucial for finding ideas. The most promising hypotheses regarding differences revolved around the balancing of values.
Conclusions There is inarguably a palpable cultural difference between Rust and Go programmers. It was possible to find evidence for different traits of programmer culture with only a limited understanding of the humanities and limited time. A collaboration of researchers from software engineering, psychology, and sociology could yield significant improvements over this research project.

Name:	Busch-George
Vorname:	Leon Martin
Studiengang	Mono-Bachelor Informatik
Matrikelnummer	4663608

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Datum: 30.04.2023

Unterschrift: Leon M. Busch-George

Contents

1 Research Objective and Approach **7**
- 1.1 Introduction . . . 7
- 1.2 Goal and Approach . . . 7
- 1.3 Related Fields . . . 8
- 1.4 Culture in Software Engineering Research . . . 8
- 1.5 Choice of Communities . . . 11

2 Reflections on Methodology **14**
- 2.1 More Successful Methods . . . 14
- 2.2 Less Successful Methods . . . 19
- 2.3 Rejected and Unattempted Methods . . . 21
- 2.4 Relevant Considerations . . . 22

3 Cultural Differences **23**
- 3.1 Safety . . . 23
- 3.2 Community . . . 25
- 3.3 Simplicity . . . 28
- 3.4 Hacker Culture . . . 31
- 3.5 Fundamental Values and Pragmatism . . . 33
- 3.6 Undeveloped Ideas for Differences . . . 35

4 Results **36**
- 4.1 Methodology . . . 36
- 4.2 Differences . . . 36
- 4.3 Outlook . . . 37

1 Research Objective and Approach

1.1 Introduction

Software engineering involves solving "wicked" problems [1]: The initial situation and the target solution can have many interdependent variables that change over time and the requirements might not all be known or even contradict each other. There could be multiple legitimate solutions of which neither is technically correct - or no solution at all. Determining which information is relevant can already be a highly contentious matter. Agreeing on the potential outcomes of a design decision, formatting guidelines, or a proposed change is often equally difficult. From understanding the requirements to deploying the solution, software engineering is far from being as deterministic as it may be perceived to be. A piece of code or a software design document also has properties beyond what is relevant to the behavior of the results. Both can adhere to or deviate from a convential form, inspire confidence in the reader, or provoke concerns - not much unlike a speech or a letter. With the more universal instructions for producing high quality results being less specific, there is often room for different viable methods - much as there are no rules that guarantee the creation of impactful poems.

On the one hand, having "no definitive formulation, no stopping rule, no right or wrong, and no immediate or ultimate test of a solution" [47] adds to the complexity of software engineering. On the other hand, this degree of freedom allows for the expression of attitudes, beliefs, values, conventions, goals, desires, and aesthetic appeal - traits that can be understood as being 'cultural'. This research strives towards a better understanding of the more cultural traits of programming work and how to compare different groups of programmers in terms of their culture. The goal is to provide follow-up research with ideas for detecting and reasoning about these traits in hope of bolstering a notion of culturally diverse groups of programmers rather than monistic cultures on the base of value-judgements.

1.2 Goal and Approach

This bachelor thesis paper is a report on a small-scale explorative research project with the aim of finding cultural differences between groups of programmers, determining the concepts that are meaningful for the differentiation and, ideally, easy to operationalize in follow-up research, and documenting the means for obtaining them. Candidate hypotheses were tested to create a preliminary assessment regarding these criteria. Methods for obtaining and testing hypotheses were decided upon ad-hoc with the aim of achieving a balance between reasonable effort and sufficiently suitable results. An evolution of specific phrasings of research questions was used to guide the research and make reasoning about potential methods and findings easier. A central phrasing was determined to be: "Inhowfar are there systematic differences in the way that Rust programmers and Go programmers approach the solution of problems - most importantly in those cases where there are few or no guidelines from the framework itself?" Over the course of the research, the sensibility for cul-

[1] Selected aspects from different interpretations [45].

tural differences was intended to increase. The Rust and Go programming language communities were chosen for this study. The decision is covered in detail in chapter 1.5.

The main part of the research was intended to repeat the following steps:

1. Acquire data like software development artificats and public communication to develop hypotheses
2. Revise hypotheses using interviews
3. Reflect on methods and findings

This iterative study approach was inspired by grounded theory (GT). The researches sensibility for cultural differences was to increase over the course of the research.

1.3 Related Fields

Despite the research objective being rooted in software engineering, there is a strong focus on behavior and group dynamics which belong to the disciplines psychology and sociology. In the domain of computer science, it falls under the broad definition of Behavioral Software Engineering (BSE) proposed in 2015 [30]. There is also a small overlap with the field of Human-Computer Interaction (HCI) which applies behavioral questions in a software context - but usually to users rather than developers.

1.4 Culture in Software Engineering Research

There are two important underlying assumptions to this research: *1.* programming languages are linked to ways of thinking and *2.* software culture correlates with software community association.

'Linguistic relativity' is a theory that describes an effect of natural languages on cognition with increasing success [46]. This idea is not often applied to programming languages. Notably, Venkataraman et al. call for research on "how language and symbols [..] affect emotions of software engineers" - emotions, in turn, influencing "imagination and creativity" which are seen as being important to the "craft"" of software engineering [44] and, in a blog post from 2017, a polyglot programmer reasons about their behavior being influenced by the programming language [2].

The second assumption (software culture being related to community association) is trivial on one hand because of the interpersonal aspect of culture 1.4.1. On the other, an important aspect of software communities should be considered to better appreciate the idea of researching their culture: Software-related communities are usually open for participation in governance and relatively easy to join and leave (more so than, for instance, a sports organization). That sets them apart from other culture-related communities that are, for instance, based on territory, family relationship, or - to some degree - work environment. With little available external means for binding members, programming language communities are highly reliant on other means -

[2] https://blog.chewxy.com/2017/09/20/sapir-whorf-programming-languages/

like distinguishing themselves through values. This was believed to facilitate a shift of research methods: The choice for a community would be a more important expression of culture and less attention would need to be paid to language rules predetermining the expression of culture through programming behavior. Thus, asking study participants about their community preferences becomes an additional way to reason about culture, next to the more established analysis of software development artifacts or reasoning with participants about programming decisions. At the same time, this poses a new requirement for choosing participants and, to some extent, the choice of software community. For example, a software community comprised mostly of programmers who are paid extra to keep alive old infrastructure would express their culture through community association to a lesser extent - at least not in a way that is relevant to software engineering practices. Under these considerations, it was assumed that noticing cultural differences between members of different programming language communities would be possible. A closer inspection of the mechanisms of mutual influence between individual and community values and practices was not considered to be otherwise important at this stage of the research.

1.4.1 Relevant Literature concerning Culture in SE

Culture was found to be a multi-faceted concept even outside the software field. Fazli et al. named a few common aspects: "Culture is learned, culture is associated with values and beliefs and behaviors that are shared by a group" [10, p. 2]. In their paper, they also described how they searched for and filtered papers relating to culture in the context of software engineering. They have manually inspected and classified over a thousand papers that were published between 2000 and 2017 and found 20 matches that covered collaboration and international culture. That approach was not deemed to be feasible for this research project because of the amount of work required.

Like with Fazli et al., a broad search was conducted on the ACM Digital Library and the IEEE Computer Society Digital Library for this research because of their key roles in the field of computer science research. Additionally, the Web of Science was used to look for papers with a high quantity of citations, hoping they could be relevant in finding an established understanding of culture in software engineering, and provide hints for research methods. Most notably, Sharp et al. argued that there is evidence for a 'culture of software engineering' and that it is important for the profession to understand this culture [37]. They did so using methods from ethnography and psychology, such as performing discourse analysis on conference material, conducting interviews, and surveying the work practices at multiple companies. Another prominent effort is that of Lenberg et al. aiming to establish the field of 'Behavioral Software Engineering' [30]. They have also performed a more systematic literature review.

1.4.2 Rejected Categories of Culture in Literature

Most publications in software engineering mentioning the world 'culture' fall into one of these three categories:

- Shaping organizational culture and aligning goals to improve the quality of a

software product

- National differences of programmers in international teams adversely affecting the work
- The impact of personality traits on Computer Science Education

The most visible works fall into the first two categories. Their concern with culture is embedded in an organization context and tailored towards increasing work quality or efficiency by 'controlling' cultural factors through governance - or education. In a way, that line of thinking opposes that of this research. Rather than postulating a central culture for programmers to follow, the intention is to cultivate a pluralistic understanding. This does not mean that the underlying understanding of culture is incompatible with this research: In a publication from the first category, the term 'culture' is used to describe, among others, "enduring values, norms, attitudes, competencies, and behaviour" in an organisation [11]. The issue with the use of such work in this research lies within their bias. The aforementioned publication is a book about security engineering that treats culture as a means - "culture must be continuously fostered", "awareness must be developed", and "procedures must be enforced" [11, p. 334]. Other goals from the first category are the reduction of the need for knowledge transfer or of friction when re-assigning staff. The results often are recommendations for laying out codebases in the form of reusable and self-contained modules, avoiding 'goto' statements, or establishing standards for code formatting, deployment technology, and problem models.

The second category was also understood to be biased towards organizational values. More and more companies are opening up to the internationalization of their workflows - not only with software. And so, an increasing effort is being put into establishing "practices, images, notations, standards and tools that help actors to shape and align their work, so that it becomes readable and usable by relevant others" [39, p. 71] (i.e. across the cultures of programmers in different countries). The understanding of culture is, again, not entirely inapplicable in the context of this research project. With the works being prompted by problems that occurred with international collaboration, the reports of cultural differences and the methods for testing them were expected to relate primarily to national culture.

The primary concern with the last category, 'education', was that it was seen as being inherently biased towards traits that the authors or the teachers deem relevant during teaching and learning. Another concern was that students typically do not yet know much about programming and are unlikely to have already developed a strong cultural identity regarding software practices. It was anticipated that there could be some research in which the authors or the teachers showed awareness of student's preferences relating to programming - as there is for preferences regarding learning methods [32]. No such works were found during the literature research before it was given up out of time concerns.

All three categories were considered to be only tangential to this research. While they might apply a compatible concept of culture, their findings and methods were not believed to be appropriate because of the inherent bias towards external values or their being based around presupposed cultural differences. Additionally, work

and education research often treats culture as a means to end rather than as a phenomenon. Effects related to the desire to stay in an employment relationship or related to herd behavior were seen as being likely to skew findings towards behavior and values that individuals would not have exhibited otherwise. To an extent, work and education settings are also a subject of choice but the effect can vary dramatically and the degree of choice (or lack thereof) would have to be adjusted for.

1.4.3 Other Considerations

Hacker Culture 'Hacker culture' is an established term even outside software-related research. The customs and values commonly attributed to hacker culture are not specific to software engineering and they are also practised by people who do not engage in software development. But they do relate to software engineering and appeared appropriate for basing hypotheses on. Such aspects could be security-awareness, excitement from overcoming boundaries, creativity, cleverness, or the inclination towards reappropriation.

Data Mining using Development Tools In other research regarding software developer behavior, measurements are taken from the IDE, for instance using plugins that collect usage data. This method is not entirely appropriate because workflows can vary systematically between frameworks. Indeed, workflow-related differences should be expected when comparing Go and Rust programmers as both platforms come with their own tooling. Another argument against this kind of testing is that the study participants would be required to use a specific tool which could affect their behavior or their willingness to participate. For hypotheses regarding tools or specific metrics that might be less of a problem but for open-ended questions the skew was thought to be too difficult to control for at this scale.

1.4.4 Conclusion

There was evidence for the recognition of the importance of research into the human factors of software engineering in the the papers that were considered more closely during the exploration of literature. Little evidence was found for an established open-ended approach on the term 'culture'. Many papers contain methods or ideas for methods that were estimated to be useful for this research but many of them also seem to entail a risk of biased results.

1.5 Choice of Communities

Technology-agnostic Software Communities The biggest and best-known software communities, like Stack Overflow, GitHub, or 'r/Coding' on Reddit could be thought of as obvious choices for a comparison. They are very big and assumedly diverse, as they are almost completely agnostic towards problem domains and software frameworks. Their data is very accessible with some of them also providing special interfaces for researchers. Despite the importance of these communities for activities related to software engineering, potential measurable differences in community-specific behavior were assumed to depend more on the rules of communication or the trust

and reward mechanisms of the platform than to values more closely related to programming. Paired with the assumption of language-culture relatedness 1.4, members of these generic software platforms were expected to exhibit behavior that would be difficult to compare and interpret regarding the research question.

Go and Rust Go and Rust are, in many ways, very similar phenomena. Both are openly-developed system languages that were backed by well-known software companies while they gained popularity. Their development was mostly sponsored by Google and Mozilla A.8 [3]. Their similarities are not surprising, as both aim to solve a variety of problems that C++ is prone to [4]. Among those problems are error-prone concurrency and memory management models, unpopular build systems, inconsistent approaches on documentation, limited dependency management, or competing and incompatible standard library implementations. With their integrated build systems, tools for formatting, documenting, and deployment, Go and Rust are not only programming languages but software development platforms - Rust even has a central package repository [5]. Both languages name safety as important goals: "Simplicity, safety, and readability are paramount." (Go) and [16] "Safety in the systems space is Rust's raison d'être" (Rust) [21]. They value correctness and deterministic builds and both link statically by default.

Development on Go started in 2007 and the work on Rust started in 2006 [6]. Version 1.0 of Go was released in March 2012. Version 1.0 of Rust was released in May 2015.

According to the "Stack Overflow Developer Survey", Go and Rust programmers have similar salaries [7] and both languages are represented similarly in work contexts [8] [38]. Interestingly, the data from that survey indicates that people working with Rust are not interested in using Go and vice-versa which indicates that there is significant difference between groups.

Despite their similarities, Go and Rust 'felt' different enough to base this research on them. Rust is found to 'look' more like C++ and Go more like C. Rust has built-in metaprogramming macros (that can be used to implement something resembling runtime reflection), Go has built-in reflection. Rust is commended for having "a panoply of [language] features with something for everyone!" [29, ch. 19] and has only a basic standard library. The inventor of Rust said: "If the language is only good at one thing, it'll be a failure." [2]. On the other hand, "Go does not compete on features." [34] while having a more ample standard library, most of which is related to network applications.

[3] Mozilla dismissed a significant amount of Rust developers in 2020 and the community has set up a highly-federated foundation. Go is still strongly moderated by Google.

[4] For Go, Cox et al. name such problems as being related to C++ [8]. Its invention was sparked by slow compile times with C++ [15]. For Rust, Hoare - the inventor - says in an interview: "Our target audience is "frustrated C++ developers"." [2] noting in his personal blog that "[he does] not mean to pick on C++" [21]. Jung et al. associate the problems that Rust solves with C++ (among others) [28]

[5] The Rust community's crate registry: https://crates.io

[6] The earliest commit in the Go repository with a plausible date is from March 2008; though it appears the early git history of the project was the ground for experiments. For Rust, there is a separate repository with commits from before 2010 (https://github.com/graydon/rust-prehistory).

[7] 89.204$ p.a. for Go and 87.047$ for Rust

[8] 12% for Go and 9% - not exclusive

Perl and Python Another language pair that was considered for this research project is Perl/Python. Both were invented in the late 1980s and are object-oriented and 'multi-paradigm' interpreted languages. And both are concerned with human consumers of code, albeit in very difficult ways The language pair was seen as having a relation that is similar to that of Rust and Go. Python has a "Zen of Python" that touches upon similar values to that of "Go Proverbs": Clarity, beauty, readability, and the ability to reason about code. Perl is more open to adding features and supporting different usage paradigms.

A key downside of the Python-Perl language pair was seen in the communities' demographics. The Perl community is significantly smaller than that of Python. Being a Perl programmer was suspected to be the result of a selective mechanism much more so than being a Python programmer would. Another concern with this language-pair was the differences in the ranking mechanisms of PyPI and CPAN (the respective package repositories) because it was expected before the research project that code analysis would play a bigger role. Also, Perl's culture relates strongly to hacker culture, the reappropriation of existing tools, and artistic and playful expression. Paired with the small size of its community, this was understood as further increasing the risk of reducing the expression of culture 1.4. The finding that Perl developers being paid significantly more than Python programmers [38] can be seen as indicative of an inherent non-programming-related bias. In combination with the overall high pay of Perl developers, this could also be evidence for Perl developers being highly sought-after mainly for maintaining 'sunsetted' projects.

The assumed difficulties with comparing Perl and Python could be addressed with an appropriate study design but it was decided that it would be too laborious for this type of research. Go and Rust were deemed to be easier to work with for analyzing and reasoning about software communities.

2 Reflections on Methodology

The purpose of this section is to give an overview of the methods that were used for and/or deemed relevant to finding, deriving, and assessing hypotheses in this explorative research project. The results were not intended to hold up under scrutiny or undergo null-hypothesis significance testing, which is typically expected from a research paper. Rather, the results should provide grounds for reasonable arguments and ideas for follow-up research.

2.1 More Successful Methods

2.1.1 Publications of Software Development Platforms and Platform Authorities

Publications by platform authors and official handbooks and guidelines had already been used for orientation in preparation for this research project. There were two principal concerns about using these resources: *1.* Platform authors and maintainers are not representative members of the community and *2.* their assumedly high influence over community behavior could be seen as governance. Originally, it had been intended to carry out more interviews. But a failure to procure the expected amount of interview partners made a change in the approach necessary. Therefore, time that had become available went into the analysis of platform communication instead. The principal research question 1.2 highlights the importance of avoiding the interpretation of behavior that programmers had not engaged in freely. This constraint was intended to apply to, for instance, effects like Go programs using fewer functional paradigms than Rust programs - Go does not support functional programming in the way Rust does. The causal influence of, for instance, the "Zen of Python" over programmer behavior is different from that of language grammar. The constraint could also be seen as applying to rules enforced by companies but both the companies' affiliation with the programming platform and the programmer's affiliation with the company can be considered expressions of culture themselves and, thus, mediate the effect - as was argued regarding the programmer's relation to the platform 1.4. With this more relaxed interpretation, behavior exhibited by programmers that is in alignment with platform authors' values would still be seen as expressions of culture 1.1 under most circumstances.

The lines of reason provided by platform authors (e.g. official documentation, conference material from authors) and the importance of values were, in fact, found to be mostly consistent with those expressed by platform community members in the interviews. The more strict interpretation of the constraint would likely entail comparing Rust and Go programmer behavior regarding writing a different language like Python to rule out the effect - regardless of where the hypotheses were drawn from. Other than informal publications from the platforms themselves, personal blogs of people who have some form of authority over either platform were also found to be useful [9]. With the assumed connectedness of an individual's culture with that of the platform 1.4 supported by the interviews, platform values were seen as providing reasonable grounds for hypotheses. The origin of these hypotheses has to be docu-

[9] For example: [7] [23] [42]

mented so that they can be tested accordingly - for instance by letting Go and Rust programmers solve related problems using Python.

"Go" and "Rust" are common words that also used extensively in other contexts. This was a recurring difficulty while searching for research papers or online resources. A way to address this is using the more specific terms "Rustlang" or "Golang" but these terms exclude relevant results from search results - more so for Rust than for Go.

Platform Resources In a way, the previous line of reasoning also applies to platform documentation. For the most part, handbooks or specification documents on the language and the platform itself promote values around their usage. A notable difference is that the authors of Go have established a set of 'proverbs' around the usage of Go - similar to the 'Zen of Python'. At times, Go's specification and language documents also express disapproval of values. This results in Go being viewed as an 'opinionated language'. For instance, the specification of the Go memory model reads: "If you must read the rest of this document to understand the behavior of your program, you are being too clever. Don't be clever." [40]. This played into the considerations for the 'Fundamentalism and Pragmatism' 3.5 class of differences.

Apart from this aspect, platform resources were not found to show striking differences. A systematical analysis was considered to be promising regardless, especially around the weighting of values (e.g. explaining how Rust's borrow-checker avoids dangling pointers while 'politely' avoiding any opinion on the matter).

Software Community Resources Public discussions on software community platforms were also considered to be interesting for investigations on ideas and hypotheses. The advisor for this thesis paper had pointed out constraint for discourse on Stack Overflow (SO). The rules rule out questions that are subjective or unprovable In an attempt to initiate public discourse for feedback on hypotheses, an open-ended question about decision-making was asked on SO. The question had been written in awareness of the rules, giving examples for proof of behavior differences and value weighting, and was deleted regardless. Hacker News on Y Combinator [18] is not limited in this regard but is also more closely related to hacker culture. The restriction also does not apply to 'r/Coding' on Reddit but that has not been attempted because the platform is not structured towards systematic answers and was thought to be more prone to biased results and misleading answers [10].

A type of communication that has not been considered more closely is learning forums where programmers can ask questions about their code. The people asking there might be new to programming or learning another language. These channels are likely mainly useful for posting questions to test hypotheses.

Analysis of Conference and Teaching Material Community conference and teaching material might be less susceptible to the assumed bias in teaching research 1.4.2. Resources on portals like slideshare.net are difficult to rank because of their small

[10] Answers can be ranked highly for other reasons than correctness or relevance, like humor. The ranking depends on the Sub-Reddit's moderation as well as Reddit's guidelines.

size. Youtube provides an extensive collection of platform resources and offers transcripts of videos than can be seen as being more representative. The analysis of slides, videos, and transcripts using discourse analysis or GT methods might be viable but they were not attempted because of a lack of ability and knowledge.

2.1.2 Interviews

Artifact analysis in code and communication had been found to be difficult without strong guiding hypotheses 2.2.2. Interviews were selected as the primary means for obtaining qualitative data and directions. An important limitation of interviews is that interview partners are not always aware of their reasons and, if they are not, the brain is good at filling in such gaps using anything it deems plausible [11]. Field notes were chosen as the method of documentation for their ease of access [12]. The loss of information and the inherent bias were deemed to be acceptable under the premise that the sensibility of the researcher is also a subject of the research 1.2. The aim was to reach average programmers using Go and Rust, preferably in a team. The information acquired from the prior community research was the foundation for interview topics. A questionnaire was set up to be used as a guide for a semi-structured approach A.4. The questions were changed and extended to adjust to new findings on differences and reflections on methodology. One phase of the interview was used to ask open-ended questions about their stance on the platform and the community. In the other phase, participants were asked more directly about particular aspects of community, values, and the presumed relation to their - or the respective other - platform. This was intended as a means to test hypotheses and to gather hints for differences in value and behavior.

Candidate referral Two candidates who went through with the interview were proposed by the advisor of this thesis. A third candidate who had been referred by a work contact agreed to the interview but then did not respond to the email asking for an appointment or to the reminder a week later. At a later stage, another candidate was referred by a former colleague and they agreed to answering questions in a chat, but they showed a substantial lack of programming knowledge and their entry was discarded. One researcher offered to refer students as candidates for Rust interviews. That offer was ignored because of students generally being less likely to have developed distinguished notions of programming and identity. Also, the researchers' work group topic was related to safe programming with Rust which was seen as a problem since the referred students might have been particularly biased. Ultimately, three referred candidates were asked to participate and two completed the interview.
Personal and work contacts were found to be very hesitant to ask their peers and colleagues about participating in the interview. Quality and availability of candidates depends very much on the source of the referrals.

[11] Extreme examples of this being split-brain patients confidently providing false explanations for a behavior using the half of their brain that contains the Broca area despite the other half being solely responsible for the behavior or healthy patients explaining their alleged motivation for a behavior that was triggered by an electrode.

[12] Taking field notes does not require consent and is significantly more reliable than audio recordings.

Work contacts At least seven work contacts were asked to participate, of whom two agreed to the interview. Candidates were selected to not be in an active work relation with the researcher. The successful interviews were conducted with a contractor and a consultant.
An advantage of this method is that it is easy to know the programming language used by candidates. A clear downside is that there is not necessarily much interest to respond to a work contact for spending time after work hours [13].

Personal contacts Five personal contacts who were assumed to be using Rust or Go were asked to participate and/or recommend other candidates. Of the personal contacts, one agreed during an informal conversation and the interview was conducted immediately. Four of them did not see themselves as being Rust or Go programmers. Two of those made comments about either language via email and in a chat conversation. This mode of recruitment also depends on personal contacts for the quality of the entries and the ability to find participants. If both parties have already allotted time for a conversation and find the venture to be interesting, it can be very easy to set up an interview [14]. Regardless, even limited success was not anticipated and is unlikely to reproduce well, so this is not a recommendation. As with work contacts, those affiliated with the researchers are not appropriate participants for the production of more rigorous results.

Public Call for Participation While only four interviews had been completed and two other interviews were still pending (one of which ultimately failed to take place), requests for participation were posted to various online channels. Requests to promote the interview in digital meetups were sent to six meetup organizers on meetup.com, none of whom responded. Emails were sent to four key members who were active in their respective community and at conferences. They also did not respond. Given the impression gained from the previous interviews that Rust developers were enthusiastic and liked to talk about Rust, it was initially considered to be surprising to get no responses at all. Later, this was determined to not have any meaning regarding the hypothesis because of the small sample size and success depending heavily on the contact request itself.

When these attempts failed, requests were posted publicly to communities on Discord and Matrix along with the questionnaire containing the base questions for a possible interview. There was a respondent agreeing to an interview who turned out to be a recruiter and another response with a filled-in questionnaire was submitted by a community member with a science and teaching background. This last submission was not an interview but their response included the desired arguments and they agreed to contact via email and supplied complementing information.

Public calls for participation were assumed to be slightly more risky than the more personal approaches - the reason being a supposed stronger selection mechanism for participants with more vigorous opinions on the research topic.

[13].. or with an old colleague ..
[14]Also, it helps immensly to know which programming languages your friends and family are using.

Approaching Random Commnunity Members A handful of community members were approached on chat platforms (on Discord and Matrix). They were chosen at random from public rooms. They turned out to be interested in the topic and responded to informal questions but none agreed to an interview with voice communication.
This approach was attempted for longer than was probably reasonable without producing utilizable data. Even after abandoning the approach, it appeared that it should be possible to find a way of setting up the conversation that would more likely lead to an interview. The partners in the conversations appeared to lose interest after being asked for an interview. Maybe they felt tricked. In hindsight the chat conversations were a missed opportunity: Chat partners might have given permission for their chat responses to be used for research - instead, only interviews were asked for. One chat partner rejected the interview because his spoken English was not good enough - in hindsight, the added coverage is a benefit of interviews in written communication. This attempt was believed to have had a beneficial effect regarding the sharpening of sensibility for community differences.

Evaluation Conducting interviews turned out to be a useful method for investigating possible cultural differences. The more structured first phase of the interviews confirmed a general alignedness of individual and platform values. The second phase turned out to be both very fruitful and rather delicate. The suggestive effect of questions was deemed to have been underestimated during the first interview. Addressing the participants' sensitivity to suggestive questions was attempted by updating the interview questions for the first phase accordingly [15]. For the second phase, a list of potential questions was added to the questionnaire. Letting participants talk without any guidance might be a good way to avoid unintentionally provoking an answer but unguided monologues produced only mixed results and were not pursued further after the second interview. For the second phase, participants should ideally have expressed their position on a subject before they were given a question regarding their reasons. It was decided that missing an opportunity to ask a question was preferable to skewing the feedback because, around that time, it was becoming evident that interviews were more difficult to obtain than had been anticipated.

Agreeing on dates for interview appointments was laborious which made planning difficult [16]. Interview partners were carefully selected to ensure a diverse pool of participants. The restrictions on interview acquisition should probably have been relaxed more quickly to increase the availabilty of interviews for the evaluation of hypotheses in the earlier stages of the research. At first, work contacts were asked to recommend colleagues for the study. This was intended to produce a well spread out body of participants but this early attempt failed to procure candidates in the desired way. With only two interviews thus far, personal and work contacts were asked to participate themselves but most of those who answered refused, due to not matching

[15] E.g. 'Why do you like X?' was changed to 'What do you like about X?' because it implies X being liked. 'Is there anything you like about X?', while being an even more passive phrasing, slightly implies the opposite. 'Why are you using Rust?' was not considered because 'why' can include circumstantial reasons or a request for justification which might risk triggering 'filled-in' answers.

[16] It took 8 and 9 days (respectively) and 6 emails each to set up the successfull interviews with the referred contacts. For work contacts, it took 7 and 16 days (respectively) and 7 emails.

the target profile When it became evident that no more interviews would result from these contacts, participation was requested on community platforms. The early effort that has gone into representation considerations was likely disproportionate given the lack of study participants.

The acquisition of demographic background information was not part of the initial version of the questionnaire and was added afterwards. The information missing from previous entries was provided on inquiry. All participants were working in teams with other programmers of their group (Go/Rust). All except two participants were employed programmers. The other two had founded their own companies creating Go and Rust products, respectively. Two participants (one in each group) had less than a year of experience using the respective language. The participant demographic was considered to be appropriate for a small-scale project like this. Given the overall difficulty of obtaining interview partners, this was seen as a success. At the same time, the careful selection of partners turned out to be overwhelmingly time-consuming. That is likely something that can be addressed by involving more experienced researchers from the human sciences.

Apart from the shortcomings in the acquisition of interview participants, the interviews presented an oportuniy to ask questions about programming in general - knowledge that might have proved useful. This was only realized afterwards. The focus on platform-related hypotheses during the preparation for interviews resulted in various platform-related ideas but more general questions about preferences and customs would also have been hugely beneficial to the overall research goal. Chat conversations and submissions of questionnaires might complement interview data for future studies as well for their other qualities - like being more accessible for programmers who do not speak a mutual language well enough. Combining these methods could also help accomodating different preferences for communication. The filled-out questionnaire sent in via email was deemed sufficiently similar to the interview responses and was treated like one. Accompanying the interviews with submissions from publicly spread questionnaires could be useful for testing hypotheses as it is easier to prepare and requires less effort than setting up and conducting interviews. In order to deal with preferences for setting up appointments, a suggestion is to both propose a date for the interview in the initial outreach and to provide a tool for setting appointments at the same time (and stating that either one is acceptable). A customer relationship management tool (CRM) might also prove to be useful, especially when using more than one communication channel or dealing with larger numbers of candidates.

2.2 Less Successful Methods

2.2.1 Showing Participants Code of the other Platform

The second Go interview partner agreed to be shown Rust code and, when shown a code sample using lifetimes and traits A.10, said that they were unable to understand the code. The code sample was intended to be used for the collection of general feedback first and then, maybe, for a brief discussion about memory management and means to interface with it. An attempt was made in explaining the code but it was aborted because the interview partner was not comfortable with learning the

related concepts during the interview [17].

Both the time requirement for explaining the code and the implications for the interview quality were deemed good reasons not to pursue the approach. Rust programmers are likely more capable of reading Go code than vice-versa. Go programmers who understand Rust in a short-enough time are probably proficient enough with Rust that it becomes important to find a better distinction of Rust and Go programmers 2.4.1.

Ideas for Improvement It should be possible to address the issue of Go programmers being less likely to be able to understand Rust code: One obvious way might be the selection of code samples that are easily understandable. That might only be possible for certain hypotheses as the code samples matching this criterion are likely already biased towards specific patterns and, thus, likely to produce no or skewed results. Another way around could be using a third languages that is likely understood by both groups. These ideas were considered late in the research and were left unattempted. Code samples could also be created using suitable hypotheses in the language of the platform that the participant is identifying with. The difficulty of this method lies in the careful crafting of code samples. Again, it can only be used with a limited number of hypotheses.

2.2.2 Quantitative Measurements on Software Development Artifacts

During the early stages of this research, it was anticipated that code analysis could be among the more important means for deriving or confirming difference hypotheses. This assumption was not nullified but, instead, slightly complicated: In order to find expressions of culture in artifacts of behavior, the cultural traits of which the behavior could be an expression of must be determined first. For instance, the addition of two integers 'a + b' can be interpreted in a variety of ways: Regarding clarity of the code, one might find that 'a' and 'b' are not very descriptive names. Regarding run-time safety, it could be said that there is no protection against integer overflows. Because of the openness of the interpretation it was deemed that this approach would be less feasible for finding hypotheses.

A collection of software projects from comparable problem domains was compiled for both manual inspection and systematic testing A.2. It was agreed upon that the approach was feasible despite the sample size effectively being one for each domain. Indeed, there were differences regarding some metrics for one domain that a second project was pair added to [18]. As a consequence of this finding, more project pairs have been added. Rust's and Go's main language repositories are also included but they are treated separately. They do not compare well with other software projects in many regards because of their difference in size and scope. All software projects are FOSS which can be seen as skewing the results. Follow-up research projects might be able to obtain source code access from a large enough sample of closed-source projects.

[17] They reported after the interview that they had learned the purpose of the lifetime annotation.
[18] For instance, the 'container' project pairs behaved differently regarding the 'far merge' metric A.5.

2.2.3 Asking other Researchers for Help

Two psychologists at Reichman University in Israel and Oregon State University in the United States were contacted to ask about ideas for testing a hypothesis regarding the testing for cultural traits and regarding adolescent behavior - they did not respond. In retrospect, this was likely due to the fact that the requests were rather unusual and the idea of comparing the behavior of grown-up professionals to that of adolescents can easily be discarded as being insincere.

A publication in which code recognition timings were measured [25] was sparked by a bachelor thesis [24]. The author of which proposed two concepts for differences that were deemed to be promising in the context of this research 3.6.2 3.3.2.

They also proposed FPI-R as being an 'economic' method for testing personality traits. With the test allegedly taking from 20 to 30 minutes and target traits being only weakly related to programming, this was rejected for the moment but it might become relevant for future research when better proven methodologies are required.

2.3 Rejected and Unattempted Methods

2.3.1 Survey

Surveys are typically used for collecting quantitative data. A web survey tool could have been used to get a few quick answers to questions that pop up. Also, it might have resulted in a larger amount of submissions. A proper mechanism and tool would have had to be either found or developed. Members of the likely more privacy-oriented communities might have objected to links to Google Forms or SurveyMonkey. An attempt was made to create a web survey tool that would allow the distribution of a survey link to communities while communicating a privacy-friendly use of the data. This idea was abandoned. While survey methods might have been adapted to fit this research, they were rejected. Typical considerations going into surveys were deemed to be misguiding because they tend to revolve around quantitative acquisition of uniform data. The questionnaire that was used in the study was oriented more towards interviews and qualitative data.

2.3.2 Noteworthy Ideas from Literature

The research of Sharp et al. involved different kinds of research methods and multiple companies [37]. Research methods were chosen in such a way that it could accomodate for the "nature of the particular collaboration". This likely relies on having some form of relation to the company but, at the same time, might be very insightful.

In a CSE context, Tomer et al. used GT to investigate how Indian computer science students developed their professional identity in terms of 'identity morphing' [41]. They conducted interviews over the course of six months and they were able to develop and validate hypotheses.

2.4 Relevant Considerations

2.4.1 Classification of Participants

Since determining traits of Rust and Go programmers was the object of the research, no such traits could be used to distinguish Rust from Go programmers. Instead, participants were simply asked to identify as either. Requests for interview participation were sent either directly to candidates, some of whom were known beforehand for their platform-association, or published on community communication platforms, from which the platform-association can reasonably be inferred. Still, candidates and participants who lacked experience with the respective platforms were excluded.

2.4.2 Avoiding 'Contaminated' Traits

Similarly to the problems when selecting study participants to test hypotheses on, the selection of hypotheses should take into account the possibility that traits can have non-software-related origins rather than software-related ones. For instance, a fairly high portion of Go programmers comes from China and China has significant scores in three of the six dimensions from Hofstede's cultural model. Testing for such traits might produce seemingly positive results that are not causally related to platform differences. That is, of course, unless the aim of the research is to test the link between local (national) culture and that of a platform. Zoluorrati et al. were able to confirm such correlations using data from Stack Overflow [48]. For follow-up research, it might be possible to either work around such issues or incorporate them, but it is out-of-scope for this project.

3 Cultural Differences

This section is concerned with the differences between the Go and Rust platform communities expressed by programmers. For each hypothesis, there is a short, tabular overview containing a preliminary assessment and ideas that are hoped to be useful for follow-up research.

3.1 Safety

Safety - the "freedom of accidents" [1, p. 16] - is one of the central drivers behind Go and Rust 1.5. Interestingly only one participant in each group mentioned 'safety' directly as something they liked about using Go/Rust. For Go, this was with the ability to easily identify safe code; for Rust, it was part of a list of what makes Rust dependable. Instead, interviewees mentioned 'stability', 'dependability', 'ownership rules', 'type system rules', 'community mindset [for producing dependable code]', 'formal proofs' (Rust) and 'frugal design', 'static typing', 'good error-handling', 'clear and consistent code', 'integrated tests' (Go). They all relate to safety, which is seen as supporting evidence for the assumed language-community relatedness.

So far, this is a similarity; the differences lie in how safety is achieved. In a way, Go aims to attain safety by being a simple language and encouraging users to write clear code. Rust prefers "mechanism not policy" [20] for providing safety while using its many features. For instance, Rust requires the use of an 'unsafe' block for changing global variables [19] - Go does not and leaves checking for races from simultaneous data access to the programmer. With Rust, the compiler makes programmers aware of many types of safety issues. With Go, this is mostly up to the runtime and to readers of the program code.

Difference	Safety
Assessment	No indication of a difference in value. Rust programmers are likey more aware of certain classes of safety.
Follow-up notes	Evidence for the awareness of different classes of safety might be expressed in Go code more so than with Rust.

Safety and security are entangled values [1, ch. 28]. It is noteworthy that, while Rust is closer related to safety, Go has a stronger focus on security [20] and it was only mentioned by on Go participant.

Difference	Security
Assessment	Go developers might be more concerned with attacks on their software.
Follow-up notes	Maybe related to 'harmful software' 3.4.

[19] Rust acknowledges that this is not always possible: "People are fallible, and mistakes will happen, but by requiring these five unsafe operations to be inside blocks annotated with unsafe you'll know that any errors related to memory safety must be within an unsafe block." [29, ch. 19]

[20] The heading of the website reads: "Build simple, secure, scalable systems with Go" [13]. Go has built-in tools for fuzzing and a 'crypto' package inside the standard library that can be used for TLS out-of-the-box.

3.1.1 Reliability

According to Anderson, assurance and sustainability are the most difficult topics in security engineering [1, ch. 28] - they are concerned with "whether the system will work, and how you're sure of this" (assurance) and "how long it will keep on working" (sustainability). 'Reliability' was chosen as a combination of these terms in an attempt to find differences regarding safety values. Rust places reliability prominently on its website: "A language empowering everyone to build reliable and efficient software." [35].

As with safety, reliability was not mentioned directly during the interviews. Go aims to be reliable by being easy to maintain and review, which in turn is achieved by writing programs that are easy to understand [21]. This was mentioned by one Go programmer. The Rust participants mentioned 'dependability' (an aspect of assurance), the idea that Rust programs would "still run in twenty years", and 'stability' (both classified as 'sustainability') as something they liked about Rust. The platform stance for Rust is 'mechanism not policy': The compiler and the language generally prohibit unsafe behavior. This was also was mentioned only once ('type/memory/thread safety').

Difference	Reliability
Assessment	No indication for a difference in value.
Follow-up notes	Try to operationalize the importance 'correctness' and 'understability'.

One aspect better highlighted by reliability than by safety is sustainability and maintainability. As with the other safety aspects, Go is about simplicity and easy-to-understand code. Rust participants mentioned an alleged longevity of Rust programs - Rust itself even promises to never remove old editions of the language from the compiler. This indicates a difference in the means for achieving sustainability goals that could be tested further.

With further understanding of this difference, it might be possible to come up with more specific hypotheses and prepare code snippets or similar artifacts to ask developers about. For example, selections of Rust code could be shown to Go programmers for asking whether they would merge it.

Difference	Sustainability
Assessment	No noticable difference in value.
Follow-up notes	Could Rust developers think that that the context of their programs wont change?

[21] "Readable means Reliable" [34]

3.1.2 Example Follow-up Considerations

Various aspects of sustainability could be considered. From the Go perspective, writing code that is more difficult to maintain might be a detriment to 'reliability growth' - the idea that a system's dependability increases slowly over time while bugs are fixed [1, ch. 28].

How often do Rust developers intend to revisit their code? How do they treat the idea that inactivity of a project is often considered a detriment to security rather than a sign for the absence of bugs?

In April 2023, a renowned security engineer described a bug in the crypto package of the Go standard library [43] . The bug was a regression from switching to a constant-time function that did not include a bounds-check. For Rust, there exists a crate [22] that allows the authors of functions to require preconditions that a caller has to confirm they are upholding. Would such a solution be interesting for Go developers?

Non-Software Considerations Hofstede et al. define two "dimensions of national culture" that are related to reliability: "uncertainty avoidance" and "long-term orientation" [26]. The model has been argued to have only limited applicability in a software context by Fazli et al. [10, p. 8] (among others) and testing for traits like these requires special attention 2.4.2. Both 'dimensions' are defined beyond the common meaning of the words - some aspects of which might be interesting for follow-up research. It might be possible to distinguish programming communities using Hofstede's model but it has to be kept in mind that such differences might be caused by the geographical distribution of communities more so than by motivations from software engineering.

3.2 Community

Rust and Go are open-source and openly developed platforms but they depend largely on their sponsors. The share of commits made by project-associated authors in each main language repo is at least 40% for both platforms A.8. They care immensely - but differently - about their communities. Both acknowledge from early on that a flourishing community is important [23].

For Rust, the community code of conduct was set up specifically to make members who are sensitive feel more welcome [22]. Rust should 'empower' programmers by giving them 'superpowers'. The compiler is meant to be "friendly" and the error messages it produces "useful" [24]. One of the interview partner named the welcoming nature of the community as being one of the reasons for them joining Rust; another said they enjoyed the friendliness. The Rust community is notacibly diverse. Throughout the previous years, Rust has been the most 'loved' programming language in the Stack Overflow Developer Survey [38]. Maybe the strong focus on friendliness could have an impact on the companies that are adopting Rust?

[22] https://docs.rs/pre/latest/pre/

[23] "We look forward to collaborating with the community to make it a successful one." [12]. The first two entries on the Rust blog close similarly [36].

[24] "Rust has [..], a friendly compiler with useful error messages [..]" [35]

Go intends to be welcoming by being easy to learn and to understand. All interviewees mentioned ease of use when talking about what they enjoyed about Go. "We don't know where the next great idea will come from. We need all the help we can get. We need a large, diverse Go community." [6] While Go is not as 'loved' among Stack Overflow survey participants, it is slightly more used than Rust (12% to 9%). Its popularity on Github is much more pronounced [25].

Having traced out the role of community for the respective platforms, a possible hypothesis is the reverse: How important is their platform to programmers? The last two interview partners were asked why they would not use their platform (Go/Rust) for a new project. The response for Go was: "use the right tool for the right job". For Rust, a set of "constraints" was described that would result in another language being used. The wording might indicate that Rust is a default choice for Rust developers, while Go is just an option for Go developers 3.5.

Difference	Passion for Platform
Assessment	Pre-existing data favors Rust. Confirmed by interviews on small scale.
Follow-up notes	Ask about language choice process. Maybe find finer metrics for 'passion'.

Still, the adoption of Go is much higher than that of Rust. There could be economic reasons for that. Maybe companies are hesitating because of the higher learning curve for Rust. That would have been difficult to explore during the interviews because all participants were professionally active with their language. To better grasp the possibility of individual differences, it might be helpful to investigate the impact of company preferences on the higher adoption of Go.

Difference	Importance of Platform
Assessment	Pre-existing data favors Go. No conclusion.
Follow-up notes	Different pool of participants required. Maybe decision makers in companies. Alternatively, ask more developers about how they use Rust/Go and about their reasons for using other languages.

All interviewees were asked about their thoughts on working with other programmers of their respective platform. The first Go respondent said he did not have enough experience (having worked as a Go team lead for 3 months). The second Go respondent said he did not think the other members on their team considered themselves Go developers, but they were getting along well. While answering a later question, they said they did not feel the urge to belong to a community. Their use of the word 'community' was arguably different from that in the research question but it points towards a difference that does not include a sense of belonging which is otherwise seen as being important to 'community'. The third Go respondent talked about the structure of the community and the quality of packages rather than their experience with other developers [26]. This response is indicative of a view on community that

[25] 259k Go repositories, 58k Rust repositories (forks excluded). Search results from 2023-04-28

[26] Matching more closely the sentiment of "Community on Cargo" in the third Rust blog entry https:

is not necessarily liked but appreciated for other reasons. All Rust correspondents responded positively, saying they enjoyed the open-mindedness, the friendliness, or the diligent work.

Difference	Liking of Community
Assessment	Rust developers liked their community. Go developers avoided answering the question about what they liked about the community.
Follow-up notes	Maybe the deviating answers communicate 'Why is my liking of the community relevant?'. There are signs for less 'emotional' forms of appreciation.

Both platform have established the importance of community from early on, with Rust being more focused on friendliness. Another interpretation of the interview answers might be that Go programmers view the community very professionally. A further investigation of behavior and concepts relating to the importance of community was hindered by by a lack of knowledge. As a way around, the 'professional' interpretation could be found in other aspects as well. For instance, there is a proverb concerning the code formatter 'gofmt': "I don't like how it formats, but I really like that it formats" [33]. 'gofmt' also does not have to be liked to be appreciated.

Difference	Nature of importance of community
Assessment	Slight indication that community, for Rust, is a symbol for belonging and friendliness and, for Go, more of a tool for sharing code and ideas.
Follow-up notes	Other interpretations possible, like 'Rust developers are more empathic' or 'Go programmers view the community as a professional environment.'

Another way to grasp this difference could be considering the nature of the community association using the name of members. Go programmers are called 'Gophers' and Rust programmers are called 'Rustacean' in the respective communities. All Rust programmers said they were Rustataceans while none of the Go programmers said they were Gophers. This was originally a filler question in the interviews because it was not expected to lead to a difference. One Go participant mentioned a possible reason for their hesitation to call themselves 'Gopher': Allegedly, the 'retarded gopher' was a symbol for the clunkiness of early versions of Go and the later adoption of the gopher was seen as self-mockery. The other Go participants did not mention the 'retarded Gopher', one of whom confirmed to have not known about it at the time of the interview.

Difference	Identification with Community
Assessment	Go developers use the platform-provided identity less.
Follow-up notes	Larger study necessary. Risk of skew by e.g. asking at meetups or conferences.

`//blog.rust-lang.org/2014/11/20/Cargo.html`.

3.3 Simplicity

Simplicity is a core concept of the Go programming language design. Source code written in Rust should be simple as well. Rust does not seem to be as worried about more-difficult-to-read code. 'Simplicity' is mentioned only twice in the Rust Book [29] (excluding choices regarding code examples). One of the occurences hints at the idea that simplicity generally not more important than performance [27]. This is seen as contrasting with the Go proverb "clear is better than clever". There are places in the Rust Book where ease-of-use and readibility are valued more positively: "References are a complex feature, and one of Rust's major advantages is how safe and easy it is to use references." and "one long line is difficult to read, so it's best to divide it." [29, ch. 2] None of these occurences are very prominent. For Go, "Simplicity, safety, and readability are paramount." [16].

Two Go interview partners brought up simplicity of code during the interview. One Rust interview partner was asked about the perceived complexity of Rust code and remarked: "Rust has a different form of simplicity".

Difference	Simplicity
Assessment	Valued more in Go's communication.
Follow-up notes	It might be worth asking more Rust programmers about what makes Rust simple to use for them.

3.3.1 Complexity

Two measures have been chosen to quickly get a reasonably reliable assessment of complexity: Indentation of source code [28] and far merges in Git A.5. Of the example projects, Rust source code is indented farther than Go source code in every categor except one. The Rust repositories also tend to have a higher proportion of far merges than their Go counterparts A.5. The Rust repository also has significantly more far merges than the Go repository.

In attempt to make sense of the difference in nature of the importance of community for programmers, the codes of conduct were inspected more closely A.3. Not much has been identified regarding the community but the editing of the text was seen as confirmation for a more general applicability of this hypothesis regarding simplicity. It could be that Rust developers are less bothered by complexity in general. Interestingly, there exists a Rust crate for quantifying the complexity of code [29].

[27] "In this circumstance, giving up a little performance to gain simplicity is a worthwhile trade-off." [29, ch. 12]

[28] Using `tools/similar_project_file_stats` from the project repo A.1. The tools makes other measurements as well.

[29] https://docs.rs/complexity/latest/complexity/

Difference	Acceptance of Complexity
Assessment	Rust developers could accept/trust complex solutions more than Go developers.
Follow-up notes	The Rust compiler itself is much more complex technology than that of Go.

In regard to cognitive requirements and accessibility, Go is fairly straightforward: "Don't be clever" [40]. Rust, on the other hand, is commonly considered to be difficult to learn and difficult to understand. This was reflected in the interview partipants: One Go interview participant was a Go team lead just three months after starting to use Go. One Rust participant pointed out that it took him rather a long time to learn Rust. Interestingly, the inventor of Rust said that they thought writing Go programs had a higher cognitive load despite the language itself being less complex [30].

One Rust participant reported that they liked being 'forced into a relevant pattern of thinking' and that Rust would allow them to focus 'purely' on the problem at hand. Given that most problem domains are usually not concerned directly with program internals like memory management and performance, this likely confirms that a reduction of the experienced difficulty is possible through the internalization of language concepts. On the other hand, the *forcing* 'into patterns of thinking' implies that Rust can still have a high cognitive demand even for experienced programmers who have internalized them to a stronger degree. This remaining demand could indicate a limit to the internalization - or something else entirely. Testing the experienced cognitive demand will likely require the careful selection (or crafting) of code samples using concepts that participants are familiar with in order to find possible differences in levels of cognitive aptitude rather than knowledge.

Difference	Cognitive Complexity
Assessment	Rust developers could be more willing or able to process large amounts of information.
Follow-up notes	At its core, this is a psychological concern. At a later stage, identifying complexity within source code might become relevant [25]. Hansen et al. offer a different approach [19].

Fun R. Pike implies on a slide that making the language more readable or more complex involves "a key tradeoff: More fun to write, or less work to maintain?" [34]. One Rust participant said they valued the "freshness" when working with other Rust programmers. Maybe Rust developers have a conflicted opinion on complexity because they secretly enjoy the challenge of working with complex code? There is another relevant interview contribution: "[the compiler is] annoying enough for me to implement only the most fitting solution". This could be interpreted as both challenging and supporting the hypothesis. What being annoyed at the compiler turns out to be

[30] "Go's a good language. It's less complex than Rust, but also less ambitious. [..] Rust statically differentiates all these cases, divides memory and pointers up into different types, which means we can control safety and performance much better, but at the cost of the programmer having to think more." [2].

depends on the context. In the context of a game, it could be an important contributor to enjoyment.

Difference	Enjoyment of complexity
Assessment	Mixed results. Maybe Rust developers view programming as more of a game.
Follow-up notes	There might be a difference between actual enjoyment and excitement about writig complex code (also comes up in 'hacker culture' 3.4).

3.3.2 The Notion of Code

Different notions of code could be the cause for varying requirements in cognitive complexity. This differentiating concept was submitted by Hofmeister who also prompted the idea: *code should be concise and generic (code is math)* and *code should be comprehensible, readable, and specific (code is language)* [31]. The associated properties ('concise', 'readable', ..) are highly subjective and dependent on other values. The difficulties for non-Rust programmers to read Rust code were seen as more likely to be related to the unfamiliarity of concepts like borrowing, lifetimes, traits, and macros. Ultimately, it was decided against using the *language/math* differentiation because it was not believed to lead to a significant and meaningful differentiation of culture in a reasonable amount of time (untangling related values etc.).

One interview partner, who was overseeing a software company, noted that there would be "less TODOs in Rust code" and that Rust projects could be considered 'done' at some point - at least more so than with other languages. This was seen as the reason for a potential hypothesis regarding the human consumer of source code losing relevancy over time. An evaluation of the example projects did not reveal any obvious pattern regarding the amount of TODOs. Comparing the standard libraries of the languages, 0.09% of the lines contained TODOs for Go and, interestingly, Rust had no TODOs at all. This hypothesis was also abandoned.

'Code conciseness' was considered to the most promising idea for a distinction but generating a preliminary assessment was found to be very difficult because of the differences of the languages (Rust offering a lot more features to maximize conciseness). It was decided to ignore conciseness unless it was brought up by a participant (which it was not).

3.3.3 Explicitness and Control

Another concept that can be seen as contributing to the complexity of Rust code is that Rust requires programmers to be very explicit about things like ownership, mutability,

[31] 'Code' and 'language' should not be interpreted literally in this context. Ivanova et al. have had programmers read code while conducting neuroimaging and determined that the language system responded "weakly or not at all" to source code [27]. Instead, the multiple demand system was activated, but not necessarily in the same way as it is when solving math problems. Also, They found different activation patterns from reading ScratchJr or Python but there are likely easier methods for reasoning about different software cultures than brain scans.

declarations, lifetimes, or types. In general, the Rust compiler only infers information if it can do so unambiguously. Go is very similar in that regard - the difference being that Go does not have as much to be explicit about (limited support for generics, no lifetimes, no ownership). Few select things are implicit in Go like the default values for unitialized variables, default types for constants, or interface implementations.

This is seen to come up often during learning where the artifacts of Rust's explicit nature confront learners with concepts they are not familiar with. It is also seen as being related to the control of low-level program behavior.

One Rust participant said they liked the explicitness of Rust, another said they liked that Rust made them aware of dependencies of objects. No Go participant mentioned explicitness and one said they like that Go made them aware of API boundaries. Expressiveness was not mentioned at all.

One of the Rust participants also noted that Rust allowed them to focus on the problem they were solving. That was seen as supporting the idea that Rust developers indeed want to have control over ownership, lifetimes, and mutability. In Go on the other hand, many aspects of concurrency or memory management are handled automatically in the background.

Difference	Low-level control
Assessment	Mixed results. Rust is more focused on low-level programming.
Follow-up notes	This might turn out to be selected for by the platform and not the community per-se.

Other considerations Perhaps there is a difference regarding making programs to measure by stripping the unnecessary and focussing on the essential. What is essential and necessary depends much on the context and the individual programmer. Should it be possible to find these kinds of 'minimalists' in both groups, it might be helpful to ask them about their reasons for being there.

3.4 Hacker Culture

There are different conceptions of hacker culture. Wikipedia names a few common aspects [17]. This list contains a derived set of values that might be useful to this research:

- creating and sharing software
- individual freedom
- rationality
- cleverness
- intellectual excellence
- playful approach

In part, these values also relate to those of free and open source software which will have to be considered when engaging with these communities in order to test hypotheses. The aspect of cleverness appeared to be particularly interesting in regard to Go's inclination towards clarity (rather than cleverness).

Difference	Hacker culture
Assessment	Slight indication that Golang is preferred by hackers.
Follow-up notes	Hopefully easy to test using discourse analysis.

Harmful Software One participant who advocated 'the use of the right tool for the right job' named writing security-related software as being a use-case for Go. They argued it was useful for writing proof of concept for exploits as well as defending against network attackers. Indeed, searches for related software projects on Github produced less results for Rust than they did for Go A.11 but that could be caused by different levels of maturity of related features in the platform. However, that delay in the features being implemented could reflect a lack of interest in the community.

In order to properly test this hypothesis, a deeper understanding of both the culture and community of actual hackers would be necessary. Also, it would have to be considered inhowfar the communities of intrusive hackers and those of practicing security experts and enthusiasts overlap. It might well be that a typical Rust programmer enjoys finding flaws in programs or systems but refrains from harmful use of software. The information that has been found and compiled alone does not suggest that the interest of Rust developers and Go developers on this topic diverges significantly [32]. All interview participants were asked what kind of project they would use Rust or Go for and no one else mentioned software security.

Interestingly, the description for a book called "Black Hat Go" [33] contains a noteworthy statement right at the start: "Go [..] revered by hackers for its simplicity, efficiency, and reliability". The description for its Rust counterpart "Black Hat Rust" [34] closes like this: "Rust is the long-awaited one-size-fits-all programming language [..] thanks to its unparalleled guarantees and feature set". Both relate to the complexity family of hypotheses.

Difference	Harmful software
Assessment	Inconclusive. Go probably closer related than Rust.
Follow-up notes	Need additional knowledge.

[32] At one time, the Golang runtime has found its way into the databases of various signature-based malware detecion products A.12. Allegedly, that was because many of the malicious sample submissions included the Go runtime but the researcher does not remember how that conclusion came about and did not find a suitable source for the claim.

[33] https://nostarch.com/blackhatgo

[34] https://kerkour.com/black-hat-rust

3.5 Fundamental Values and Pragmatism

At multiple times throughout the research, there was a desire to capture the strength of beliefs regarding the platform. The terms 'fundamental' and 'pragmatic' were found to revolve around this idea. The common interpretation of the term 'fundamentalism' relates specifically to religious movements [31]. Here it is applied more broadly and in a more literal sense (i.e 'essential', 'foundational'). The goal was to capture how much (-> pragmatic) or little (-> fundamental) a group was willing to compromise on values.

However, this quickly led to problems. As an example, the match and switch clauses of Go and Rust are considered: The Go compiler will accept a program without a default clause - a match clause in Rust, on the other hand, must cover all cases the compiler deems possible in order to compile. Given the desire to reduce undefined behavior at runtime, requiring programmers to define behavior for unexpected values is a 'pragmatic' choice. The same requirement can be seen as being 'fundamental' for letting this decision get in the way of writing simple and readable code. This differentiation does not work without a fixed frame of reference because otherwise the balance of values can change arbitrarily (like in the example). When applying the term 'fundamental', the frame of reference is usually implied - as is a supposed qualitative inferiority. In effect, this is prone to produce biased statements and unfounded accusations, which is likely the reason why the *Associated Press* advises against using the term for groups [35]. An upside of this was found to be that both terms can be used regardless: As signals for different weighting of values in a specific context. When such a signal is encountered, it can prompt a related question about the person making the statement: *Inhowfar is there a difference between the speaker and the subject (of the term) that the speaker thinks the subject is not able or willing to compromise on X in a situation Y?* The concept 'fundamentalism' came up in the contexts *safety* and *platform purity*. The differentiation of Rust and Go programmers based on the general interpretation was abandoned. The updated understanding leads to the method being very similar to the overall approach on differences (focusing slightly more on how programmers make compromises rather than comparing the resulting decisions.).

Safety For instance, neither Rust nor Go regard regard safety as an absolute. Go allows dereferencing a pointer without a prior check which can lead to segmentation faults. With Rust, the compiler rejects any code it considers as leading to 'undefined behavior'. A resource must be either owned or borrowed for it to be used and no segfault should occur from the language perspective. Still, there are safety compromises when using Rust. In Go, 'unsafe' is the name of a package intended for the circumvention of type safety features. In Rust, 'unsafe' code blocks allow access to features that can compromise memory safety. Interestingly, the documentation calls these 'superpowers' nonetheless. The inventor of Rust called the idea "pragmatic safety" - a "device for applying (or ignoring) social pressure." [20]. Evans et al. found such compromises when asking Rust programmers about their reasons for using 'unsafe'

[35] "[.. has taken on] pejorative connotations [..]. In general, do not use *fundamentalist* unless a group applies the word to itself." Sadly, no copy of 'The Associated Press Stylebook' could be acquired. The quote is taken from a secondary source [3].

blocks: “the Safe Rust alternative is too verbose or complicated (25%) [..] and [it is] faster to write code with Unsafe Rust (5%)” [9].

Because of the platform difference, programmers of either language make safety compromises in different places and at different levels which makes a comparison difficult. The use of the ‘unsafe’ keyword in both languages may be a starting point. This is seen as also being intrisically coupled to the ‘simplicity and complexity’ family of hypotheses: Go programmers accept the risk of crashing at runtime as long as they are allowed to write what they think is a more readable program.

Difference	Safety compromises
Assessment	No reasonable data.
Follow-up notes	Go programmers have more room to make safety compromises. Compare reasons for compromises.

Platform Purity In the context of rewrites, an interview partner confirmed the (assumedly reasonable) assumption, that Rust programmers do not generally think that all programs should be written in Rust. There is an ongoing effort to remove the dependency on Python while bootstrapping the Rust compiler [36]. This idea might be considered ‘not very pragmatic’ by Go programmers. A more recent issue is about ‘purging’ Python altogether [37]. While general considerations are being voiced, it is notable that the idea is not dismissed entirely: “I don’t think this is worth spending contributor time on.”, “I think only once it’s clear what python actually can be rewritten should we actually discuss whether it should be rewritten.”.

For Go, there has been a similar discussion about moving from bash to POSIX sh [38]. The immediate respone being “Would this solve any actual problem? I don’t think I’ve ever seen anyone complaining about make.bash being a bash script in the past 6 years.” and the issue was automatically closed for inactivity [39].

This might also relate to safety (maybe the inclusion of other languages into the Rust ecosystem causing worries about safety). There is also a ‘pure Go’ sentiment that could be related. Rust developers could be asked about using Go Assembly (a generic, intermediate assembly language) because of its usefulness for supporting architectures. They could be asked about adding Python build dependencies to their projectst.

Difference	Platform purity compromises
Assessment	Go developers are less dismissive of dependencies on software written in another language.
Follow-up notes	Go developers might be less worried by the language of a build dependency but how about linking against Rust code?

[36] https://github.com/rust-lang/rust/issues/94829
[37] https://github.com/rust-lang/rust/issues/110479
[38] https://go.dev/src/bootstrap.bash
[39] Automatically closing issues could be seen as ‘negligence’ from a Rust perspective.

3.6 Undeveloped Ideas for Differences

Other ideas regarding potential differences were developed until a point where no more information could be obtained without a reasonable effort. Or they were not tested because of time constraints. The following ideas were deemed worthy of mention.

3.6.1 Drive To Rewrite Existing Software

'Rewrite it in Rust' is name given to the phenomenon of unwelcome issues requesting a rewrite in Rust [4]. It is unclear inhowfar this happens more with Rust than it does with Go A.9. 'r/Rust' has issued a statement that they would no longer condone such behavior. A Rust interview partner said that he considered the people who opened such issues part of the Rust community as well. The 'enthusiasm' for Rust had reached non-programmers who were unable to make reasonable judgements about the rewrites.

3.6.2 Understanding of Objects

There are different ways to define the entities that are made up by data structures.

- Instruments for formal analysis vs. a means to reflect human perception (hermeneutics).
- An adaption of the rationalist vs. naturalist model might be applicable to programmers. *Rationalism* being more concerned with behavior itself and *naturalism* more with with background information. Examples for this could be imperative and functional programming (rationalist) vs. object-oriented and actor model programming (naturalist).
- composition over inheritance

4 Results

Conclusion There is reasonable evidence for cultural differences between the Rust and Go programming communities. The interview responses indicate a strong alignment of a platform's values with those of its community members. Also, there is slight evidence for a branch of the overall Rust community that is comprised of non-programmers like users and managers. Other ways of partitioning software developers into culutural communities might also be possible. Despite methodological shortcomings and a severe lack of knowledge regarding the humanities, this study is believed to be a success.

4.1 Methodology

The iterative approach has been useful for quickly finding and adjusting suitable hypotheses. The alignment of platform values and individual values allowed heavy use of platform resources to generate hypotheses. The analysis of code was not as useful for generating hypotheses but had its uses for rough assessments. With a more developed understanding of cultural differences, this probably becomes less relevant. The interviews with programmers were highly relevant for developing and advancing hypotheses because the promising differences revolve around reasons and the balancing of values. Code is notoriously bad at conveying reasonings and there was a significant difference in the degree at which Rust and Go documentation and specification covered them.

The unanticipated difficulties with procuring interviews stretched out the time betwen them 1.2. The self-imposed restrictions for the acquisition should probably have been lifted more quickly. Being able to ask follow-up questions after the interview turned out to be practical while waiting for new interviews. Structuring this report also posed an unexpected difficulty: Too many hypotheses had been considered and the findings were rather heterogeneous.

4.2 Differences

Both communities care deeply about safety and community diversity but go about it differently. Most of the discovered differences relate to values of the platform in some way. Rust relies on the language being very explicit about memory management and on a strict compiler to ensure the generated programs are correct. Go avoids much of such complexity and focuses instead on guiding programmers towards writing code that can be understood easily. This ease-of-use is likely a significant contributor to Go being used in companies. Rust focuses on an inclusive community policy, is strictly moderated, and enjoys great popularity while having a slightly lower rate of adoption. Go programmers appear to care slightly less about togetherness and more about pragmatic aspects of community like the availability of usable packages. Software security (i.e. intrusion and defense) might be more important to Go programmers.

4.3 Outlook

A larger-scale study drawing on expertise from the fields of psychology (for behavior-related matters) and sociology could yield significant improvements, especially when it comes to examining interactions between programmers. It seems advisable to conduct one or two more small-scale studies like this beforehand, to better guide such a more laborious enterprise because these smaller studies are fairly easy to implement. This report contains notes for advancing hypotheses regarding differences and finding more appropriate methods. Hopefully, raising awareness of meaningful differences helps to promote cultural pluralism and inspires searches for more ways to organize teams accordingly. "If the languages all converge, we will all think the same. But different ways of thinking are good" [34] - if we want to keep solving multifaceted problems.

References

[1] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2020. ISBN: 9781119642787.

[2] A. Avram. *Interview on Rust, a Systems Programming Language Developed by Mozilla*. 2012. URL: https://www.infoq.com/news/2012/08/Interview-Rust/ (visited on 04/17/2023).

[3] *Can anyone define 'fundamentalist'?* 2011. URL: https://web.archive.org/web/20120927095413/http://www.vcstar.com/news/2011/may/12/can-anyone-define-fundamentalist/ (visited on 04/27/2023).

[4] *Clear is better than clever*. 2016. URL: https://transitiontech.ca/random/RIIR (visited on 04/13/2023).

[5] *Code of conduct*. URL: https://www.rust-lang.org/policies/code-of-conduct (visited on 03/20/2023).

[6] R. Cox. *Go, Open Source, Community*. 2015. URL: https://tip.golang.org/blog/open-source#code-of-conduct.

[7] R. Cox. *research!rsc - Thoughts and links about programming*. URL: https://research.swtch.com/.

[8] R. Cox et al. "The Go programming language and environment". In: *Communications of the ACM* 65.5 (Apr. 2022), pp. 70–78. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3488716. URL: https://dl.acm.org/doi/10.1145/3488716 (visited on 03/14/2023).

[9] A. N. Evans, B. Campbell, and M. L. Soffa. "Is Rust Used Safely by Software Developers?" In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 246–257. ISBN: 9781450371216. DOI: 10.1145/3377811.3380413. URL: https://doi.org/10.1145/3377811.3380413.

[10] F. Fazli and E. A .C. Bittner. "Cultural Influences on Collaborative Work in Software Engineering Teams". In: Hawaii International Conference on System Sciences. 2017. ISBN: 978-0-9981331-0-2. DOI: 10.24251/HICSS.2017.056. URL: http://hdl.handle.net/10125/41205 (visited on 03/18/2023).

[11] F. J. Furrer. *Safety and security of cyber-physical systems : engineering dependable software using principle-based development / Frank J. Furrer.* eng. Wiesbaden, 2022.

[12] A. Gerrand. *Go: What's New in March 2010*. 2010. URL: https://go.dev/blog/hello-world.

[13] *Go*. URL: https://go.dev/ (visited on 04/29/2023).

[14] *Go Community Code of Conduct*. Google. URL: https://go.dev/conduct (visited on 03/20/2023).

[15] *Go: A Documentary*. 2022. URL: https://golang.design/history/ (visited on 03/20/2023).

[16] R. Griesemer. *The Evolution of Go*. 2015. URL: https://go.dev/talks/2015/gophercon-goevolution.slide.

[17] *Hacker culture — Wikipedia, The Free Encyclopedia*. 2023. URL: http://en.wikipedia.org/w/index.php?title=Hacker%5C%20culture (visited on 04/15/2023).

[18] *Hacker News*. URL: https://news.ycombinator.com/news (visited on 04/18/2023).

[19] M. E. Hansen, A. Lumsdaine, and R. L. Goldstone. "Cognitive architectures: a way forward for the psychology of programming". In: *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. SPLASH '12: Conference on Systems, Programming, and Applications: Software for Humanity. Tucson Arizona USA: ACM, Oct. 19, 2012, pp. 27–38. ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384596. URL: https://dl.acm.org/doi/10.1145/2384592.2384596 (visited on 03/18/2023).

[20] G. Hoare. *Project Servo*. 2010. URL: http://venge.net/graydon/talks/intro-talk-2.pdf (visited on 04/18/2023).

[21] G. Hoare. *Rust is mostly safety*. 2016. URL: https://graydon2.dreamwidth.org/247406.html (visited on 04/18/2023).

[22] G. Hoare. *Comment on 'Question about Rust's odd Code of Conduct'*. 2017. URL: https://.reddit.com/r/rust/comments/6ewjt5/question_about_rusts_odd_code_of_conduct/ (visited on 04/23/2023).

[23] G. Hoare. *frog hop*. URL: https://graydon2.dreamwidth.org (visited on 04/18/2023).

[24] J. C. N. Hofmeister. "Influence of identifier length and semantics on the comprehensibility of source code". Bachelor's Thesis. 2015.

[25] J. C. N. Hofmeister, J. Siegmund, and D. V. Holt. "Shorter identifier names take longer to comprehend". In: *Empirical Software Engineering* 24.1 (Feb. 2019), pp. 417–443. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-018-9621-x. URL: http://link.springer.com/10.1007/s10664-018-9621-x (visited on 04/08/2023).

[26] G. Hofstede. *Culture's Consequences: Comparing Values, Behaviors, Institutions and Organizations Across Nations*. SAGE Publications, 2001. ISBN: 9780803973244. URL: https://us.sagepub.com/en-us/nam/cultures-consequences/book9710.

[27] A. A. Ivanova et al. "Comprehension of computer code relies primarily on domain-general executive brain regions". In: *eLife* 9 (2020). Ed. by A. E. Martin et al., e58906. ISSN: 2050-084X. DOI: 10.7554/eLife.58906. URL: https://doi.org/10.7554/eLife.58906.

[28] R. Jung et al. "Safe systems programming in Rust". In: *Communications of the ACM* 64.4 (Apr. 2021), pp. 144–152. DOI: 10.1145/3418295. URL: https://doi.org/10.1145%2F3418295.

[29] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2018. ISBN: 1593278284. DOI: 10.5555/3271463.

[30] P. Lenberg, R. Feldt, and L. G. Wallgren. "Behavioral software engineering: A definition and systematic literature review". In: *Journal of Systems and Software* 107 (Sept. 2015), pp. 15–37. ISSN: 01641212. DOI: 10.1016/j.jss.2015.04.084. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215000989 (visited on 03/22/2023).

[31] H. Munson. *Fundamentalism*. 2019. URL: https://www.britannica.com/topic/fundamentalism (visited on 04/27/2023).

[32] K. Nollenberger. "Comparing Alternative Teaching Modes in a Masters Program: Student Preferences and Perceptions". In: *Journal of Public Affairs Education* 21.1 (Mar. 2015), pp. 101–114. ISSN: 1523-6803, 2328-9643. DOI: 10.1080/15236803.2015.12001819. URL: https://www.tandfonline.com/doi/full/10.1080/15236803.2015.12001819 (visited on 04/16/2023).

[33] R. Pike. *Go Proverbs*. Youtube. 2015. URL: https://www.youtube.com/watch?v=PAAkCSZUG1c (visited on 04/17/2023).

[34] R. Pike. *Simplicity is Complicated*. dotGo. 2015. URL: https://go.dev/talks/2015/simplicity-is-complicated.slide.

[35] *Rust*. URL: https://www.rust-lang.org/ (visited on 04/29/2023).

[36] *Rust Blog*. URL: https://blog.rust-lang.org/ (visited on 03/20/2023).

[37] H. Sharp, H. Robinson, and M. Woodman. "Software engineering: community and culture". In: *IEEE Software* 17.1 (Feb. 2000), pp. 40–47. ISSN: 07407459. DOI: 10.1109/52.819967. URL: http://ieeexplore.ieee.org/document/819967/ (visited on 03/14/2023).

[38] *Stack Overflow Developer Survey 2022*. URL: https://survey.stackoverflow.co/2022 (visited on 04/19/2023).

[39] H. Tellioğlu and I. Wagner. "Software cultures". In: *Communications of the ACM* 42.12 (Dec. 1999), pp. 71–77. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/322796.322811. URL: https://dl.acm.org/doi/10.1145/322796.322811 (visited on 03/19/2023).

[40] *The Go Memory Model*. Google. 2022. URL: https://go.dev/ref/mem (visited on 03/20/2023).

[41] G. Tomer and S. K. Mishra. "Professional identity construction among software engineering students: A study in India". In: *Information Technology & People* 29.1 (Mar. 7, 2016). Ed. by Hasan and Helen Henry Linger, pp. 146–172. ISSN: 0959-3845. DOI: 10.1108/ITP-10-2013-0181. URL: https://www.emerald.com/insight/content/doi/10.1108/ITP-10-2013-0181/full/html (visited on 03/19/2023).

[42] A. Turon. *turon's web site*. URL: http://aturon.github.io/tech/2018/02/06/portability-vision/ (visited on 04/18/2023).

[43] F. Valsorda. *A Cryptographic Near Miss*. URL: https://words.filippo.io/dispatches/near-miss/ (visited on 04/20/2023).

[44] Mahesh Venkataraman and Kishore Durg. "Research Idea on How Language and Symbols (Semantics and Semiotics) Affect Emotions of Software Engineers". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. ICSEW'20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 577–580. ISBN: 9781450379632. DOI: 10.1145/3387940.3392232. URL: https://doi.org/10.1145/3387940.3392232.

[45] *Wicked problem — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Wicked_problem%5C&oldid=1149475301 (visited on 04/30/2023).

[46] P. Wolff and K. J. Holmes. "Linguistic relativity". In: *WIREs Cognitive Science* 2.3 (May 2011), pp. 253–265. ISSN: 1939-5078, 1939-5086. DOI: 10.1002/wcs.104. URL: https://onlinelibrary.wiley.com/doi/10.1002/wcs.104 (visited on 04/15/2023).

[47] C. Zannier and F. Maurer. "A qualitative empirical evaluation of design decisions". In: *Proceedings of the 2005 workshop on Human and social factors of software engineering - HSSE '05*. the 2005 workshop. St. Louis, Missouri: ACM Press, 2005, pp. 1–7. ISBN: 978-1-59593-120-7. DOI: 10.1145/1083106.1083124. URL: http://portal.acm.org/citation.cfm?doid=1083106.1083124 (visited on 03/16/2023).

[48] E. Zolduoarrati, S. A. Licorish, and N. Stanger. "Impact of individualism and collectivism cultural profiles on the behaviour of software developers: A study of stack overflow". In: *Journal of Systems and Software* 192 (Oct. 2022), p. 111427. ISSN: 01641212. DOI: 10.1016/j.jss.2022.111427. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121222001327 (visited on 03/17/2023).

A Attachments

A.1 Project Repository

The git repository contains the sources for this thesis paper alongside some tools that were used to analyze projects and notes from the interviews.

https://git.imp.fu-berlin.de/lgeorge/thesis-language-culture

A.2 Example Projects

```
Container runtimes:
git@github.com:docker/cli.git (Go)
git@github.com:containers/podman.git (Go)
git@github.com:tailhook/vagga.git (Rust)
git@github.com:containers/youki.git (Rust)

DNS:
git@github.com:miekg/dns.git (Go)
git@github.com:bluejekyll/trust-dns.git (Rust)

IPFS:
git@github.com:ipfs/go-ipfs.git (Go)
git@github.com:rs-ipfs/rust-ipfs.git (Rust)

JWT:
git@github.com:dgrijalva/jwt-go.git (Go)
git@github.com:Keats/jsonwebtoken.git (Rust)

MQTT:
git@github.com:mochi-co/mqtt.git (Go)
git@github.com:bytebeamio/rumqtt.git (Rust)

Postgres:
git@github.com:lib/pq.git (Go)
git@github.com:sfackler/rust-postgres.git (Rust)

QR code:
git@github.com:skip2/go-qrcode.git (Go)
git@github.com:kennytm/qrcode-rust.git (Rust)

Readline:
git@github.com:chzyer/readline.git (Go)
git@github.com:kkawakam/rustyline.git (Rust)

Redis:
git@github.com:go-redis/redis.git (Go)
git@github.com:mitsuhiko/redis-rs.git (Rust)

WASM runtime:
git@github.com:wasmerio/wasmer-go.git (Go)
git@github.com:wasmerio/wasmer.git (Rust)
```

A.3 Codes of Conduct

A code of conduct was added to Go in 2015 [14]. The earliest reference is for the Rust code of conduct [5] that was obtainable in a reasonable amount of time is from a commit [40]. The referenced link only appears later in the year on archive.org but that is already good indication that the CoC of Rust is older than that of Go.

While the code for Go covers roughly the same behavior as the code for Rust does, it appears to spend more time rationalizing and relativizing choices. It is more structured and has a prominently placed list of values using bold fonts for readability. The accompanying blog post contains a few interesting statements [6]:

- "I believe this captures the tone we want to set, the message we want to send".
- "I haven't mentioned active exclusion based on or disproportionately affecting race, gender, disability, or other personal characteristics, and I haven't mentioned harassment. For me, it follows from what I just said that exclusionary behavior or explicit harassment is absolutely unacceptable".
- "We do not believe that all conflict is bad".

The code for Rust consists of two rather long lists with long bullet points and three paragraphs of text. Sadly, no literature accompanying the release of the code was found. Graydon Hoare explains parts of it on Reddit [22].

A.4 Questionnaire

These are the questions for the latest iteration of the questionnaire:

- What do you like about working with Go/Rust as a language? (what is important to you)
- What do you like about working with other Go/Rust developers? (what is important to you)
- How is working with Go/Rust programmers any different from working with other programmers?
- Do you consider yourself a Gopher/Rustacean?
- Where are you using Go/Rust? (work, FOSS, solo projects etc.)
- What would be reasons for you to start a project in a language other than Go/Rust?

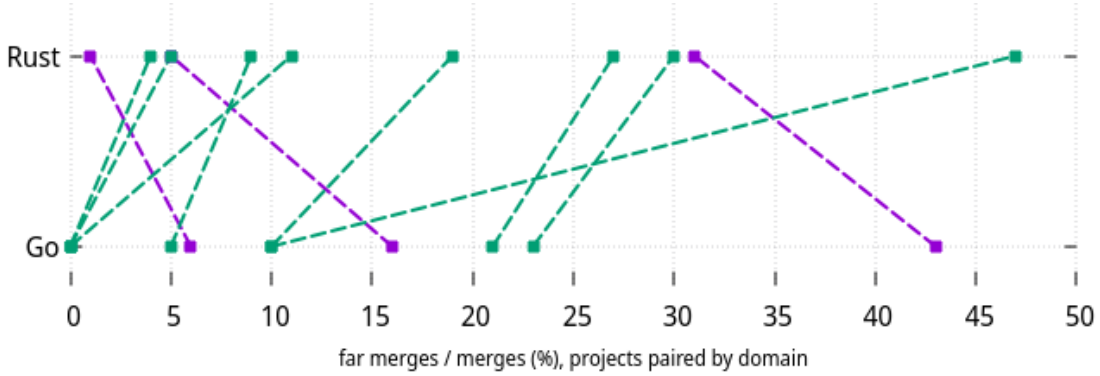
A selection of questions for the open phase of the interview:

- What is your experience with [own platform]?
- What do you make of [other platform]?
- What makes Go/Rust particularly, e.g., consistent, pragmatic, foundational, or versatile?
- Have you personally experienced a X programmer trying to convince you of using X?
- How do you decide what to implement inside your own company vs. what to include from foreign libraries?
- Go has the proverb "a little copying is better than a little dependency" - is that also true for Rust?
- Is X feature of Rust interesting for Go?
- Is programming more creation or more transformation?
- Was the addition of templates appropriate for Go?
- What do you make of Rust allegedly being very expressive?
- What makes X so popular?
- Is Go 'backward-oriented'?

[40] https://github.com/rust-lang/rust/commit/f645cad3a1557e26c3245c619a362f6dadd55e1a

A.5 Far Merges for Project Pairs

There are more project pairs for which the main development branch of the Rust repository has a higher proportion of far merges to overall merges; far merges being 'merges into other branches'.



A.6 Github User Counts

This data was generated by searching for users on Github on July 28, 2022. Two searches where conducted and the results combined to circumevent query limitations. One selecting users with more than one repository and another selecting exactly one.

Users with language ..	at least one public repo	no public repo (percentage of all users)
"Rust"	39026	2247 (~ 5,4%)
"Go"	247680	11640 (~ 4,5%)

A.7 Git Repo Metrics - Commits and merges

This data was extracted from the Rust and Go main language Git repositories using `tools/history_stats` from the research project repository A.1.
A complementary manual observation from 2021: Go has few far branches/merges. Merges accross more than 2 branches were not observed. Rust has a lot of merges. Branch depths of 8 and upwards were observed accress multiple stretches of commits.

Repository	Time	Unique commiter emails	Commits	Merges (% of commits)	Far merges (% of merges, % of commits)	Mean commit body line count
Go	May 2021	2005	48007	181 (0%)	154 (85%, 0%)	11
	May 2023	2356	56106	284 (0%)	252 (88%, 0%)	12
Rust	May 2021	4212	142913	40099 (28%)	20811 (51%, 14%)	3
	May 2023	6031	221518	66010 (29%)	42178 (63%, 19%)	3

A.8 Git Repo Metrics - Authors

This data was manually compiled in May 2021. 'PAE' is 'project associated email address', 'MAC' is a 'top ten most active commiter'.

Repository	commit authors with PAEs	commits by authors with PAE	share of commits made by MACs	MACs with project association
Go	244/2033	30518/45981	22738 (~41%)	9
Rust	47/4161	27705/138108	53657 (~39%)	10

More than half of the commits in the Go repository are created by authors with an email address matching '@golang.org' or '@google.*'. One of Go MACs does not use a Google email address and is not publicly associated with Google. Two of Rust MACs have emails from Rust or Mozilla but all are either (former) Rust team members or Mozilla employees.

A.9 Rewrite-related Projects

A quick search on Github (e.g. https://github.com/search?l=Rust&o=desc&q=%22rewrite+in+rust%22&s=stars&type=Repositories) found no significant differences in rewrite projects (44 for Rust, 41 for Go). There are about double the amount of issues regarding rewrites for Go (2.3k for Rust, 4.7k for Go).

A Reddit user asked about targets of potential rewrites on the similarly sized 'r/rust' and 'r/golang' subreddits [41]. The engagement was higher for Rust than it was for Go (490 to 111) and the post itself was upvoted higher on 'r/rust' (219 to 41), Counts updated at 2023-04-26. Sadly, the user has blocked direct communications and could not be contacted by other means.

A manual search using search engines strongly favored projects related to rewrites in Rust. This a collection of projects that were deemed noteworthy:

Rust	
Merged kick-off for in-place rewrite of fish shell	https://github.com/fish-shell/fish-shell/pull/9512
Active rewrite of GNU coreutils	https://github.com/uutils/coreutils
Abandoned port of Emacs	https://github.com/remacs/remacs
Active fork of Emacs using Rust with upstream compatibility	https://github.com/emacs-ng/emacs-ng
Active rewrite of coreboot	https://github.com/oreboot/oreboot
Go	
Abondened reimplementation of GNU coreutils	https://github.com/guonaihong/coreutils
Active in-place rewrite of the Alda backend	https://github.com/alda-lang/alda

A.10 Rust Code Sample for Interview

Code sample from RBE https://doc.rust-lang.org/rust-by-example/scope/lifetime/trait.html from 2023-02-11. The purpose was to discuss memory management with Go programmers and ask what they thought about an approach like this.

```rust
// A struct with annotation of lifetimes.
#[derive(Debug)]
struct Borrowed<'a> {
    x: &'a i32,
}

// Annotate lifetimes to impl.
impl<'a> Default for Borrowed<'a> {
    fn default() -> Self {
        Self {
            x: &10,
        }
    }
}

fn main() {
    let b: Borrowed = Default::default();
    println!("b is {:?}", b);
```

[41] Rust: https://old.reddit.com/r/rust/comments/11051xl/what_would_you_rewrite_in_rust/
Go: https://old.reddit.com/r/golang/comments/12alic4/what_would_you_rewrite_in_golang/

```
}
```

A.11 Exploit-related Repositories on Github

The repos don't necessarily contain offensive code but instead just reflect interest in the topic. This is particularly relevant for the term 'hacking'. Data from 2023-04-15.

Term	Go	Rust
black hat	79	25
red team	65	7
blue team	9	3
exploit	281	88
shellcode	92	45
pentest	182	34
hacking	1.176	526

A.12 Golang Runtime Being Detected as Malware

Examples for harmless softare being flagged by 'antivirus' software:
https://community.bitdefender.com/en/discussion/84185/false-positive-when-running-go-lang-program-edited-
https://github.com/gopasspw/gopass/issues/1807
https://forum.nim-lang.org/t/9850

This blog post includes screenshots of which product detected a 'Hello World' program as which malware: https://betterprogramming.pub/a-big-problem-in-go-that-no-one-talks-about-328cc3e71378