Mapping New Jungles: *API Jungloid Tool Generality and Development Effort Across Languages*

Group: Haoran Peng, Maddie Burbage, Tian Yu

1. Introduction

Modern software development heavily depends on static analysis-based code navigation tools. In VSCode, basic tools like “go to definition”, “type tooltip” and “find all references” are provided for mainstream languages. They are typically implemented separately using language-specific compiler toolchains. Slightly more complicated code navigation tools have also been developed to aid programmers further when working on specific languages. Some of these, like a type-based API chain explorer (or the jungloid tool, which we will elaborate later), focus on specific languages and even IDEs, without spreading further. We hypothesize that one major reason behind this is that they take considerable effort to build, given that for each language it requires learning and using a different compiler toolchain that is typically written in that language. Developing syntactic analyses that generalize well across languages is typically quite hard to do programmatically [12, 13].

Motivated by this, we aim to investigate the extensibility of one language-specific code navigation tool to gain insight into one facet of a broader question: what makes it hard to build coding tools that generalize well across programming languages? Specifically, we examine the development tradeoffs involved in transforming a nontrivial static analysis tool, originally implemented for a single language, into a multilingual and reusable form.

2. Motivation

To ground this investigation, we focus on a recurring challenge in software development: constructing an object of a desired type from available data using complex, interconnected APIs. Modern software development heavily depends on complex and interconnected APIs to create functional systems. Constructing an object of a desired type from available data might require the use of many different APIs, involving transformations across many different types. Learning how to do that in unfamiliar or poorly documented libraries can be difficult for programmers.

Consider the programming task of parsing a Rust file into an abstract syntax tree (AST) using the rust-analyzer crate, as demonstrated in the code snippet in Code 1. Despite knowing the input (String) and the desired output (ast::AstNode), programmers may find determining the correct sequence of API calls for this transformation challenging. This is because (1) it involves multiple types and methods that are scattered across crates (Rust

modules), both standard crates and rust-analyzer crates, (2) the rust-analyzer framework is rapidly changing and thus under-documented, and (3) not all functions involved are standard constructors (<Type>::new()). These context issues make it less likely for programmers to be familiar with the full sequence of API calls required to complete this task, and makes it harder for the programmer to become familiar with the required API calls. Figure 1 illustrates the graph of object transformations that programmers must mentally construct in order to successfully write Code 1.

```
let workspace_path = Path::new(&args[1]);
let cargo_config = CargoConfig::default();
let load_cargo_config = load_cargo::LoadCargoConfig {
    load_out_dirs_from_check: false,
    with_proc_macro_server: load_cargo::ProcMacroServerChoice::None,
    prefill_caches: false,
};
let (mut db, vfs, _proc_macro) = load_cargo::load_workspace_at(
    workspace_path, &cargo_config, &load_cargo_config, &|_| {}
)?;
let sema = &hir::Semantics::new(&db);
let root = sema.parse_guess_edition(file_id);
```

Code 1: An API transformation chain for constructing an AST node in Rust.

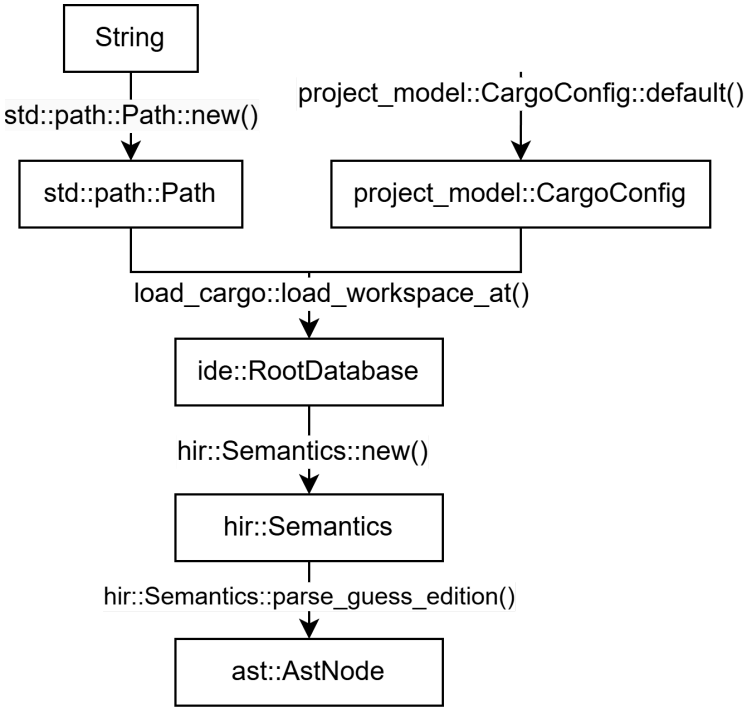

Figure 1: Graph illustrating the API transformation chain for constructing an AST node in Rust.

This challenge of discovering valid API call sequences was formalized by Mandelin et al. [1] who framed the **jungloid problem** as the task of determining a contextually valid sequence

of API calls needed to transform an object of one type into another. Their tool, PROSPECTOR, solves the jungloid problem in Java by computing necessary method calls needed to transform an object of one type into another. However, the approach has not seen wide adoption in modern development environments. Instead, programmers rely on other strategies for solving jungloid problems, rather than using a dedicated jungloid tool. According to [10], software engineers primarily rely on source code over documentation due to its perceived unreliability, often learning API usage directly through code. We believe these tools would be useful to today's programmers: we face the jungloid problem frequently in our own work, and Java hasn't changed to avoid jungloids, not to mention the fact that jungloids are present in new languages like Rust too. We hypothesize that the lack of jungloid tools is ultimately due to the development difficulty involved in constructing sound jungloid tools backed by heavy-weight compiler toolchain and IDE integrations.

We want to study the contrasts provided by a dedicated jungloid tool that is built with a different approach, where lighter-weight syntax-analysis tools that generalize efficiently across languages undergird the tool, rather than heavy-weight compiler toolchain integrations. For a tool built on these foundations, how easy is it to extend the tool across languages, and how much usability does it lose by losing some soundness and coverage of API information? This tradeoff between accuracy and generalizability affects both developer effort and user experience.

To measure these tradeoffs, we implement and evaluate Jungle-Mapper, a lightweight tool designed to automatically generate valid transformation chains between object types. Jungle-Mapper leverages modern, lightweight AST tools (ast-grep) and static analysis techniques to efficiently analyze codebases and construct transformation paths across multiple programming languages.

3. Background

When PROSPECTOR was developed with the goal of sound and powerful jungloid analysis, the available analysis and interfacing technologies made it heavily bound to a specific language and IDE. We believe the following reasons made it harder for a jungloid tool in that style to be easily replicated or extended across more settings. (1) Writing an AST analyzer required hacking a heavy-weight compiler toolchain. Heavy-weight compiler toolchains are designed primarily for code compilation and optimization, not for targeted static analysis. Modifying them to extract AST information can be complex and error-prone. These toolchains are also usually tailored to a specific language, making tool generalization to different languages difficult. (2) Early IDE integrations may have been difficult due to a lack of standardized interfaces, such as language server protocols. For instance, plugins built for Eclipse 3.x are incompatible with 4.x due to fundamental API changes. This complexity contributed to the limited use of jungloid tools. (3) The above development difficulties incentivize the creation of different tools for different contexts as demand grows for them. This requires more cost of development for the ecosystem, and higher cumulative learning curve for users to learn divergent tools.

PROSPECTOR targeted Java. As written in Code 2, in Java, creating a Timestamp object from a String representing a time requires a complex sequence of calls. The corresponding graph of transformations is illustrated in Figure 2. Programmers must navigate sequences involving intermediate types (e.g., LocalDateTime), methods scattered across different packages (java.sql, java.time), and non-constructor helper utilities (e.g., DateTimeFormatter). The issue is made worse by the fact that several valid pathways often exist (e.g., legacy SimpleDateFormat vs. modern DateTimeFormatter), requiring developers to evaluate tradeoffs between correctness, compatibility, and codebase conventions without systematic guidance. Even for seemingly simple type conversions, this ambiguity can make object construction a laborious trial-and-error procedure.

```
String dateTimeString = "1970-01-01T00:00:00";
Timestamp timestamp = Timestamp.from(
    LocalDateTime.parse(dateTimeString, DateTimeFormatter.ISO_LOCAL_DATE_TIME)
                 .atZone(ZoneId.of("UTC"))
                 .toInstant()
);
```

Code 2: An API transformation sequence for constructing a Timestamp object in Java

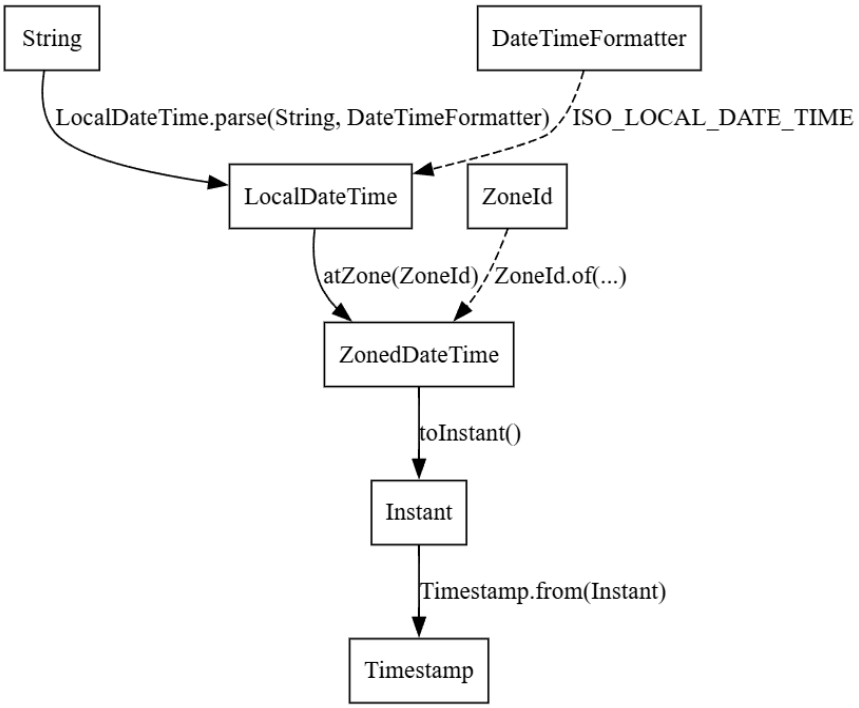

Figure 2: Graph illustrating the API transformation sequence for constructing a Timestamp object in Java

For a tool to offer useful solutions to the jungloid problem, it should be able to build a graph like in Figure 1 or Figure 2. Previously, API dependency analyses relied on heavyweight compiler toolchains specific to a language, and thus two separate tools would need to be built to produce each of these graphs from the codebase. However, the latest AST-based pattern matching tools, such as the query functionality in Tree-sitter and ast-grep, allow us to

compute these transformation chains with minimal assumptions about language semantics. This has downsides: without knowledge about the underlying dependencies and semantic differences between nodes, we will lose information. For instance, we can't perform type resolution, meaning differentiating types with names spelled the same but declared from different namespaces. We can integrate other technologies like language servers to gain this information, but it comes at a cost of developer effort, especially when extending across languages. Thus we are interested in measuring both the tool's extensibility across languages, as well as its loss in accuracy, as both affect tool usability in conflicting ways.

4. Design

We implemented our tool, Jungle-Mapper, as a static analyzer that answers user queries about how to construct an object of a specific type in a given codebase. The tool is divided into two preprocessing stages: Information Collection and Static Analysis, as well as one jungloid-query stage: User Queries.

Step 1: Information Collection. The tool preprocesses API information by analyzing source code. Because we are interested in the specific problem of querying one API's jungloid graph, we designed the tool to ingest the source code for one library at a time. After being directed by a user, the tool scans a directory containing a library's source code to determine the input and output relationships between types and APIs. This allows the tool to generate the API's jungloid graph. For this step we used ast-grep[2], an AST-based pattern-matching tool built upon Tree-sitter[6]. This library handles much of the low-level syntactic parsing required in constructing some of the jungloid relationships we need, but it does leave some language-specific pattern-matching for us to handle. Thus to search for correct and complete results using ast-grep's patterns, we need to write code corresponding to the pattern syntax required to visit a particular language's AST properly.

The information we collect in this stage includes all function and method signatures (what types they take as arguments and what types they return), the accessibility of functions (is it a public API or not), corresponding interface information, and implementation and subtyping relationships. Our design divided preprocessing into language-specific functions that use patterns to match the AST grammar of a given language, and language-ambivalent type relationship processing functions. This is because a language may have inheritance features or trait/interface-implementation relationships, but no matter what exactly the mechanism is, the relationship that allows an object of a type to be replaced with another type would be modeled as a subtyping relationship. This info can be collected by writing ast-grep patterns that match class or trait declaration nodes (which may have "extends" or "implements" clauses). A challenge here is type resolution, meaning differentiating types with the same spelt names but declared from different namespaces. Traditional heavy-weight compiler toolchains are capable of type resolution, but light-weight AST tools are not. We handled some of these cases by including namespace information in our jungloid graphs, but did not track import statements.

Step 2: Static Analysis. We then map type and API data into a graph that contains all the information that is needed for exploring any type's construction chain. There are two kinds of nodes: type nodes and API nodes. An API node takes zero or more input types (an edge from type to API) and produces one or more return types (an edge from API to type). A type may connect to another type with subtyping relationships (an edge from subtype to supertype). After constructing the graph in Python, we enter the same data into a graph database, neo4j[7].

Information contained in the graph is enough to answer the question of how to construct an object of a given type. There might be multiple ways to construct that object, and each possible way is represented as a DAG subgraph. The DAG subgraph can be constructed in the following steps: start from the target type and start backtracking along the edges; whenever reaching an API node, add all incoming edges into the DAG; whenever reaching a type node, add one incoming edge into the DAG (either subtyping edges or API output edges are okay; when there are multiple incoming edges, selecting any one of them produces a different DAG subgraph, meaning a different way to construct the target object). These DAGs may or may not stop growing and may also end up in loops, but practically it only needs to go as far as when the user knows how to construct the leaf types.

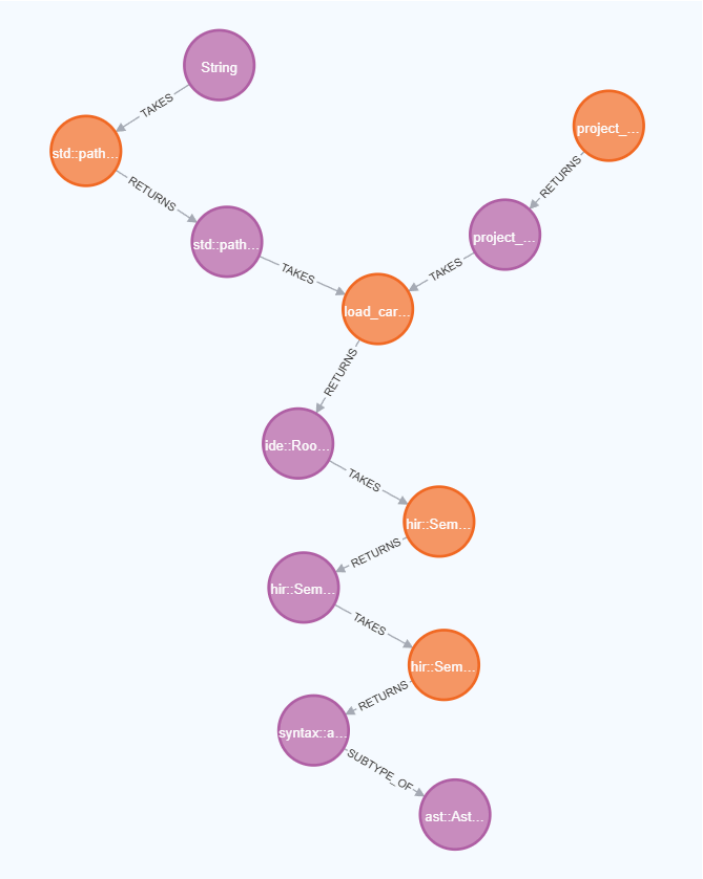

Figure 3: A graph that represents the jungloid in Figure 1. Two kinds of nodes: purple for Type and orange for API. Three kinds of edges: takes, returns, subtype_of.

Step 3: User Queries. Because our main evaluation of the tool's use of lightweight AST analysis can be performed over the tool's internal graph representations, we created a barebones frontend that is a command-line tool for querying the neo4j database containing the type<>API relationships we constructed earlier. Having a graph query language as our

graph's interface also helps us automate our evaluation over larger task sets. If the user wants to find a 1-depth DAG that constructs type ast::AstNode, a manual Cypher query (neo4j's query language) would look like Code 3. However, for longer construction DAGs, such queries that implement the jungloid static analysis logic are hard to write and require multiple iterations. Our query tool automates this process. This query tool takes one target type and zero or more source types as inputs, and produces multiple DAGs that build the target type. The provided source types are meant to be the types that the user has at hand and prefers to use, but the query tool does not promise that the produced DAGs use all or any of them, given the possibility that the user provided types cannot transfer to the target type, or that they are not necessary. Instead, the query tool prioritizes DAGs that use more of the user provided source types, but offers additional relevant options afterward.

```
MATCH (target:Type {name: 'ast::AstNode'})<-[:SUBTYPE_OF*0..]-(subtype)
MATCH (api:API)-[:RETURNS]->(subtype)  // API returns the subtype
OPTIONAL MATCH (input:Type)-[:TAKES]->(api)  // Input types -> API
RETURN
  subtype.name AS TargetType,
  api.name AS Constructor,
  collect(input.name) AS RequiredInputs
```

Code 3: A Cypher query that returns 1-depth API construction DAGs for type AstNode

5. Implementation

We implemented Jungle-Mapper in Python, and all code is available in the 776styjsu/api-jungle-map github repository. The main file handles library preprocessing, and using a simple CLI, a user can direct the program to parse a directory as a library, construct a jungle graph of type<>API relationships, and upload that to the desired neo4j database instance. We deployed this instance to our personal computers via the latest docker image of neo4j at the time of evaluation, which was tagged 2025.02.0-community-bullseye.

To interact with ast-grep and neo4j, this code also uses cypher queries and ast-grep pattern-matching rules. The neo4j configuration is handled in a separate credentials file. The parsers directory contains all language-specific analysis code, divided into separate files, to make comparison clear between different languages. Code 4 supplies an example CLI input.

```
python3 main.py ../commons-io/src/main/java/org/apache/commons/io/
java
```

Code 4: CLI to preprocess a Java library into a jungle graph stored in neo4j

We implemented jungloid queries and ranking in the queries file, which provides its own CLI. This tool accesses the same configured neo4j database and searches for paths across the graph. Tool users provide a desired end type, and optionally, any number of current start types that the tool should prioritize using in generating jungloid paths. Code 5 has an example query.

```
python3 query.py String InputStream
```

Code 5: CLI to query for a jungloid that transforms an InputStream into a String (notice the flipped order: this allows for multiple optional input types at the end)

To test our tool's functional correctness, we used the log4j 2 Java library, at version 1.2 of its API. We chose this for development because it is one of the most popular java libraries, it is large while not combining super-library combinations of functionality, and provides functionality that is replicated across many languages [8]. We tested some simple jungloid tasks against this library to spot-check and debug our tool during development, and manually collected a variety of examples of method syntax from the library to verify that our jungloid graph represented the library to meet jungloid query needs accurately. Our evaluation used separate libraries and tasks.

6. Evaluation

We designed our tool to help us investigate two facets of language extensibility when applied to jungloid solvers. First, what developer effort is required to extend the tool and how does this affect the tool's design? And secondly, how does language extensibility affect the user experience, from the interface to the expressive power of the DAG-based jungloid queries our tool runs.

Developer Effort

Cross-Language Reusability. We designed Jungle-Mapper in a way that language-specific code would be restricted to the initial Information Collection stage. It turns out that we did not need to change any of the later graph navigation or querying interface and syntax when we expanded the tool from Java to Rust. The jungloid processing pipeline kept languages abstracted out of later processing, while user query needs were based on type identifiers only, and thus worked well across both languages.

Lines of Code. Our Java-specific ast-grep processor has 147 lines of code, excluding comments and whitespace. For Rust, there were 255. In our experience, the amount of code differed due to the languages' different syntactic structures for similar semantics, especially how type identifiers were wrapped in extra modifiers, like the case of scoped_type_identifiers in Rust. Overall, this is a small amount of code but it leaves out some more complex and useful type relationships, and it requires domain knowledge and testing for comprehensiveness.

Unified Development Language and Toolchain. For both Java and Rust, we implemented the frontend ast-grep processor in Python, using the same set of ast-grep APIs. This contributes to our fast development. Traditionally, a language's AST toolchains are written in its own language, and this requires a language tool developer to first learn that language and

then learn the specific AST toolchain. However, in our case, even though the programmer who wrote the frontend ast-grep processor is new to Rust, they still succeeded in migrating the tool from Java to Rust with relatively little work.

Soundness Checking. Although ast-grep limited the amount of domain expertise needed to implement frontend processors for various languages, this still led to limitations in syntactic pattern matching where nodes in the final graph should have been merged, but were not merged, because the patterns written were not syntactically complex enough to recognize when two types were equivalent. This did not lead to incorrect answers, only missing jungloid solutions. A more egregious issue arose when certain argument types were labeled differently than others in unexpected ways, like those involving generics or arrays being labeled differently by ast-grep. Catching issues like these required a deeper understanding of the grammar in use or extensive testing. Although these issues could lead to less precise solutions, it could also lead to incorrect solutions that don't list out all required input types.

Visualized Feedback Loop. Ast-grep (and also Tree-sitter that it is based on) provides a web playground where it parses code in real time and shows an interactive AST. The AST shows the kind names and field names, and it highlights the source code section when you click on a node. This visualization greatly accelerates our workflow. For example, when a version of our analyzer alarms on an unhandled syntax node under a certain context, we would copy-paste the relevant source code section into the web playground, find out the kind name and field names of that node, observe the structures around it, and add handling for it in our analyzer. Traditionally, tool developers need to manually print out the error-raising part of the AST and check the documentation or the grammar to learn about the error, whilst this web playground provides us a much smoother development experience across both languages we target.

User Experience

We expected the techniques we chose to support our tool would trade some expressive power for ease of development across languages, so we wanted to measure our tool against a subset of the queries used to evaluate PROSPECTOR. This lets us establish a baseline compared to prior work, and we scoped the tasks selected to those with good analogies in Rust, which was the second part of our evaluation. We wanted to determine how correctness and expressiveness varied between our Java and Rust implementations, so we supplemented the PROSPECTOR queries with a set of our own manually-chosen jungloid tasks. We wanted tasks to represent popular libraries and needs within those libraries. To that end, we selected Java task source libraries from the list of most popular libraries from MVN Repository, an online Java package repository [8]. We selected a few of these that covered reasonably-different application domains and had counterpart Rust crates for analogous tasks [8].

	Java	Rust
testing	JUnit	libtest-mimic

collections	Guava	Multimap
I/O	Apache Commons IO	Walkdir, std::io, memmap2, std::path

Table 1: Jungloid task libraries chosen for our evaluation

Tasks we included from PROSPECTOR's evaluation include: Reading lines from an input stream, opening a named file, iterating over collection values, reading lines from a file, and converting a file handle to a file name. These are largely I/O and directory-based tasks, which aligns with Apache Commons IO. We add programming tasks based on some of the key API features of JUnit and Guava to cover common test setup and data structure tasks. To study similar tasks in Rust, we chose popular packages for analogous jungloid tasks based on popularity in crates.io [11]. We then translated our initial jungloid tasks from Java to Rust, discarding them if there were no analogs. One example of this discard was dropping serialization tasks performed in Gson and Serde, as there were no comparable macroless approaches to the gson tasks we'd constructed in the Rust analog libraries we examined. Task construction was an iterative process which we collaborated on through communication between one Java expert and one Rust expert.

Although we are interested in how user experience differs across languages for our tool, we are not interested in evaluating our tool's effectiveness as a user-friendly frontend, which is orthogonal to our main research questions. We instead want to understand how our tool's backend supports user queries differently across languages. To that end, our tasks represent bare-bones jungloid queries: given a specific task framing, which represents a set of desired, "correct" constructions of one output type from various input types, we measured how our tool performed.

To evaluate a task, we ran Jungle-Mapper's query tool with the requisite types and manually validated that one of the top-n returned chains corresponds with our manually-written task solution, or could produce a correct solution even if it doesn't match. We report the top correct position for the task if it is verified, and discuss our findings below.

Task	**Library**	**Input Type**	**Output Type**	**Correct Position**	**Note**
Get dynamic test executable	JUnit	String	Executable	1st	
	libtest-mimic	String	Conclusion	2nd	
Iterating over all keys in a multimap	Guava	Multimap	iterator	5th	Paths < 5
	multimap	Multimap	Iter	1st	

Filter multimap by a condition	Guava	Multimap, Predicate	Multimap	3rd	Paths < 5
	multimap			___	No similar API
Create a multimap with an initial capacity	Guava	int	ArrayListMultimap	1st	
	multimap	usize	MultiMap	1st	
Walk the current directory	Apache Commons IO	int	Stream	51st	Paths < 5
	walkdir	Path	DirEntry	1st	
Reading lines from an input stream	Apache Commons IO	InputStream	String	4th	Paths < 5
	std::io	Stdin	Lines	1st	
Opening a named file for memory-mapped IO	Apache Commons IO	String	MemoryMappedFileInputStream	___	One API node requires the relation between generic interfaces that refer to this in method signatures
	memmap2	u8	File	2nd	
Reading lines from a file	Apache Commons IO	String	List	13th	Paths < 5
	std::io			___	Path involves types declared in another crate and thus faces type resolution issues (see Limitations)
Converting a file reference to a file name	Apache Commons IO	FileEntry	String	1st	Paths < 5
	std::path	Path	str	1st	

Table 2: The Jungloid tasks with input/output types and ranking position of a correct API transformation chain

We find that Jungle-Mapper was able to successfully provide the correct answer for almost all tasks across Java and Rust. Generally, the preferred, developer-specified “correct” solution appeared as the 1st result for Rust, while for Java, the correct solution could be as low as the 51st response.

Beyond the effectiveness of our jungloid query results, we noted any language-based usability changes required by our tool. The one change we immediately needed to make concerned jungloid solution path lengths. In Rust, the libraries we used were consistently smaller than the very large Java libraries we tested for comparable tasks. This meant that jungloid searches in Rust could feasibly analyze longer chains of construction for given types, while in Java, across almost every library, the same maximum chain length would be so long that it led to path explosion during our graph searches. The Java querier and Rust querier independently tuned their path search lengths, landing on a maximum of 5 in Java, and 8 in Rust. This led to a balance between ability to find the correct solution and runtime. We also note that our querier is not optimized for usability (it returns a full list of results, rather than a human-desirable number, which may be on the order of just a few). As library size is affected by, but not intrinsic to, any language we’d consider, we think path length is an important parameter to allow configuration for, even as heuristics would enable every language’s queries to return shorter paths at the expense of comprehensive solutions over longer paths.

Finally, we are interested in comparing how the tool’s usability might change across the two languages. As we are validating a jungloid tool’s backend, and not its UI, the query structure and system efficiency are two of the most important components to evaluate. As it turned out, the query styles remained the same between both languages, with similarly sparse inputs leading to expressive and correct outputs the majority of the cases we tested.

Limitations of a Purely Syntactic Tool

Because our analysis is based on a purely syntactic tool rather than a full-functional compiler toolchain, there are certain grammatical structures that we find hard to extract information from. They are typically scenarios where the semantics of a local structure requires global knowledge to specify.

Type Resolution. When there are more than one module (essentially namespace) involved in the codebase, they might refer to different types using the same name, or refer to the same type with different namespace delimiters. An example would be that many Java libraries define their own “Node” classes, and that in two sub-crates of the serde Rust library, they refer to the same type as de::Deserialize and simply Deserialize respectively. Just by looking at a local usage of a type name does not tell the syntactic tool the full canonical name of a type. A possible solution would be having the tool collect import statements in the beginning of the file, which is similar to implementing a type resolver, and that involves more effort. Another possibility is querying an existing compiler toolchain with standardized interfaces, e.g., language server protocols.

Macros. In Rust, macros invocations make no sense either syntactically or semantically until they are expanded. Our tool does not handle Rust macros, even if they may finally expand to functions or types. In a broader view, macro is a mechanism that allows the user to define transformations from their arbitrary text to legal programs, essentially defining a DSL embedded in the host language. Traditional static analysis tools would expand the macro before analyzing, but purely syntactic tools do not have that capability. A solution is calling a compiler to expand all macros before doing the analysis. This may add to time cost but does not add too much to development time, as it is using the compiler as a whole instead of diving into its details.

7. Related Work

Besides prior work on automating the exploration of new APIs, we are interested in the comparison to modern developer strategies for ad-hoc API exploration.

Naive Text Search and VSCode “Show All References”

Manual code search is a common baseline technique for developers to explore APIs, and thus it can be used for solving jungloid problems. In VSCode for example, a developer can right-click on a method or type of interest to see code navigation options, including “Go To Definition” or “Show All References”, which provide a list of navigable, relevant lines of code within the source code. As a complementary technique, a developer can search for key terms like a method name across a library’s codebase. Both options return many results, often prioritized by the number of mentions within a file, but not by relevance to the developer’s actual motivation for the search. These strategies locate relevant library files for solving jungloid tasks, but developers must still search in those files to locate the right object construction or transformation methods to solve a jungloid task. This adds time and effort, especially in discarding false-positive results, as compared to a tool that outputs more specific methods to begin with. This strategy is widely available across languages and IDEs, so its generality and availability offsets its lack of usability in targeting or prioritization of results for the context of solving jungloid problems.

LLM-Integrated IDEs

Another current technique for obtaining jungloids centers around LLM integrations into IDEs. VSCode’s Copilot integration is an example of how developers can query an LLM and supply it with relevant source files so it generates code snippets. Programmers can prompt Copilot in natural, rather than technical language, to generate a jungloid link. They can provide a path to a codebase in Copilot’s settings to use as background context, and it can often interpret import and include statements in open files to focus attention on relevant packages. However, due to a limited attention window, Copilot doesn’t provide bounds on the accuracy or relevance of its replies on a large codebase, and it may take iterative requests to refine its answers into a useful conclusion. Static analysis tools have the potential to filter out only

relevant parts, e.g., function signatures, and feed them to Copilot in order to make the best use of the limited attention window, but currently that is not integrated into Copilot. Also, different tasks would require different context and thus dedicated static analysis. For the jungloid problem, a jungloid tool itself would be the best fit. With that extra context supplied to Copilot along with a user's requirements, it may perform better at downstream tasks such as generating code snippets according to a jungloid.

API Synthesis Tools

Researchers have developed API example mining and synthesis tools to solve related problems. In 2005, Mandelin et al. designed PROSPECTOR[1], an Eclipse IDE-integrated code completion tool to help users synthesize jungloid code for a targeted Java library. For a set of example tasks that can be expressed easily by the jungloid model, the tool performs very accurately, and successfully suggests code chains in under a second on complicated or unfamiliar APIs. However, it is targeted at one language. In one notable successor, Zhong et al. proposed MAPO[4] to generalize available Java API synthesis based on a simpler variety of queries, like by simply querying method names. Holmes et al. proposed Strathcona[9], a structural recommendation tool implemented as an Eclipse extension that automatically matches code contexts to relevant API usage examples from existing projects, helping programmers find and learn from existing usages. These tools tend to target one language at a time, and are heavily integrated with a language's compiler and IDE features. Our tool uses less optimized and accurate but more dedicated multilingual strategies for supporting API exploration.

8. Conclusion

We found some favorable signs for the present-day multilingual extensibility of jungloid tools. The jungloid comprehension problem can be effectively abstracted into layers that largely isolate language-specific development from other domains. Existing syntactic tools ease the burden of extracting many types of jungloid relationships from library source code, making developers only need to write dozens of lines of code per type<>API relationship they care to include. We noted some development downsides using this approach: notably that there are no simple soundness guarantees for language-specific frontends, leaving developers to resort to other checking methods. However, after analyzing our querying success rates, we see these as surmountable development tasks which can be tackled to slowly improve an already-usable tool in an incremental fashion. We think that the maturity of underlying syntactic tools is a large factor in enabling this development success across two languages, and would extend to others as well. The underlying structure of the jungloid problem, because it is based on type and API relationships that are key parts of many languages, suggests the rest of the tool would extend well across other languages beyond Rust. We think the ability of the jungloid problem to be broken into language-aware frontends supported by mature tooling, along with language-agnostic backends, makes this problem ripe for multilingual extension in a way that similarly modularizable problems could likely also follow. However, we believe that the number of these problems is the main limiting factor here, as it is hard to make most problems cleanly break away from language-specific

semantics while still extending across enough languages to make extension at scale a necessary question.

References

1. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. “Jungloid mining: helping to navigate the API jungle”. *SIGPLAN Not.* 40, 6 (June 2005), 48–61. https://doi.org/10.1145/1064978.1065018
2. ast-grep. https://ast-grep.github.io/
3. Lamothe, Maxime, Yann-Gaël Guéhéneuc, and Weiyi Shang. "A systematic review of API evolution literature." *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1-36.
4. Zhong, Hao, et al. "MAPO: Mining and recommending API usage patterns." *ECOOP 2009–Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings 23*. Springer Berlin Heidelberg, 2009.
5. Reid Holmes, Robert J. Walker, and Gail C. Murphy. 2005. Strathcona example recommendation tool. SIGSOFT Softw. Eng. Notes 30, 5 (September 2005), 237–240. https://doi.org/10.1145/1095430.1081744
6. Tree-sitter. https://tree-sitter.github.io/tree-sitter/
7. neo4j. https://neo4j.com/
8. mvn repository popular libraries. https://mvnrepository.com/popular?p=2
9. Reid Holmes, Robert J. Walker, and Gail C. Murphy. 2005. Strathcona example recommendation tool. SIGSOFT Softw. Eng. Notes 30, 5 (September 2005), 237–240. https://doi.org/10.1145/1095430.1081744
10. J. Singer, "Practices of software maintenance," Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), Bethesda, MD, USA, 1998, pp. 139-145. https://doi.org/10.1109/ICSM.1998.738502
11. crates.io. https://crates.io/
12. George Mathew and Kathryn T. Stolee. 2021. Cross-language code search using static and dynamic analyses. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3468264.3468538
13. D. Perez and S. Chiba, "Cross-Language Clone Detection by Learning Over Abstract Syntax Trees," 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019, pp. 518-528, doi: 10.1109/MSR.2019.00078.