Polyglot AST: Towards Enabling Polyglot Code Analysis

Philémon Houdaille, Djamel Eddine Khelladi, Romain Briend, Robbert Jongeling, Benoit Combemale

Polyglot AST: Towards Enabling Polyglot Code Analysis

Philémon Houdaille
Univ. Rennes, IRISA, INRIA
Rennes, France
philemon.houdaille@ens-rennes.fr

Djamel Eddine Khelladi
CNRS, Univ. Rennes, IRISA, INRIA
Rennes, France
djamel-eddine.khelladi@irisa.fr

Romain Briend
Univ. Rennes, ESIR
Rennes, France
contact@romainbriend.com

Robbert Jongeling
Mälardalen University
Västerås, Sweden
robbert.jongeling@mdu.se

Benoit Combemale
Univ. Rennes, IRISA, INRIA
Rennes, France
benoit.combemale@irisa.fr

***Abstract*—Today, a plethora of programming languages exists, each better suited for a particular concern. For example, Python is suited for data analysis but not web development, whereas JavaScript is the inverse. As software complexity grows and needs to address multiple concerns, different programming languages are often used in combination, despite the burden of bridging them (e.g., using Java Native Interface). Polyglot programming emerged as a solution allowing the seamless mixing of multiple programming languages. GraalVM and PolyNote are examples of runtimes allowing polyglot programming. However, there is a striking lack of support at design time for building and analyzing polyglot code. To the best of our knowledge, there is no uniform language-agnostic way of reasoning over multiple languages to provide seamless code analysis, since each language comes with its own form of Abstract Syntax Trees (AST). In this paper, we present an approach to build a uniform yet polyglot AST over polyglot code, so that it is easier to perform global analysis. We first motivate this challenge and identify the main requirements for building a polyglot AST. We then propose a proof of concept implementation of our solutions on GraalVM's polyglot API. On top of the polyglot AST, we demonstrate the ability to implement several polyglot-specific analysis services, namely auto-completion, consistency checking, type inference, and rename refactoring. Our evaluation on three polyglot projects taken from GitHub, and involving JavaScript and Python code, shows that we can build a polyglot AST without significant overhead. We also demonstrate the usefulness of the polyglot analysis services through the provided automation, as well as their scalability.**

***Index Terms*—polyglot programming, code analysis, development tooling**

I. Introduction

As modern software applications continuously grow in scale and complexity, software developers face new challenges related to lack of productivity and readability on code bases [1], [2], [3], [4], [5], [6], [7], [8]. To support the increasingly complex software engineering needs across diverse domains, a plethora of programming languages exists today. The choice of language is usually motivated from fitting it to a given problem or maximizing developer comfort. For example, Python is suited for data analysis but not web development, whereas JavaScript excels at the latter while being heavily limited in the former. As software complexity increasingly grows, combining different languages is often adopted. However, different parts of an application are not always independent and may need to interact with each other. Different programming languages are generally, by design, not able to easily mix with each other. Thus, implementing different concerns with different languages might come with its own set of incompatibility issues. This often leads to compromises in the design of an application, where a problem is not addressed with the programming language most suitable for it.

As a countermeasure, polyglot programming has emerged as a solution lately both in industry and academia [9], [10], [11]. Polyglot programming allows one to write a single program using multiple languages, with no workaround. For example, this enables the execution of code in Python from JavaScript, as well as data exchange between independent runtimes. This is different from simple language interoperability, which refers to the ability of two languages to interact efficiently. Interoperable languages can be found in several instances, such as in JVM-based languages or WebAssembly, but code may not usually be directly embed between languages.

One of the most mature polyglot representatives is the Oracle Labs' GraalVM project [11] and its Truffle interoperability framework. However, while GraalVM efforts are towards having a stable and performant polyglot runtime environment, there has been little work on polyglot development tooling at design time. There is a striking lack of support at design time for building and analyzing polyglot code. In particular, the usual programming conveniences that developers are used to in their IDEs are not yet available for polyglot code. Features such as auto-completion, type information on hover, or more complex code analysis tasks, such as refactoring or change impact analysis, lack tool support. We believe that design time support for such features is vital to boost further the adoption of polyglot programming in industry, which could revolutionize the way we develop software in the future.

To provide design time development tooling, the initial step is to build an Abstract Syntax Tree (AST) of the program.

To the best of our knowledge, there is no uniform way of reasoning over multiple languages to provide seamless code analysis, since each language comes with its own form of ASTs. As polyglot programming seamlessly mixes programming languages together in a single project or even a single file, common parsers are unable to run through polyglot code and return the Polyglot AST since they are only meant to parse mono-language programs. As retrieving the AST is the base step of many types of code analysis, this is a big obstacle to any tooling efforts for polyglot programming at design time.

In this paper, we envision enabling polyglot code analysis by means of a Polyglot AST. We present an approach to build a uniform, yet polyglot, AST over polyglot code, so that it is easier to perform global analysis. We first motivate and identify the main requirements and challenges in building a Polyglot AST with the proper qualities and we show how to implement them. We then propose general concepts for any polyglot framework implementation. We propose a proof of concept implementation of our solutions on GraalVM's polyglot API. After that, on top of the polyglot AST, we demonstrate the ability to implement several polyglot-specific analysis services, namely auto-completion, consistency checking, type inference, and rename refactoring.

We evaluate the feasibility and applicability of our contributions on a data set of three projects of 3k, 26k, and 97k LOC from GitHub that consist of Python and JavaScript code. We observe in the evaluation no significant overhead when building the polyglot AST in contrast to parsing each separate code. The various automated polyglot analysis services showed to scale well, with few milliseconds and up to less than a second of execution in the largest polyglot project.

In summary, our contributions are as follows:

1) A unified conceptual polyglot AST and its implementation to be used as a basis for Polyglot code analysis.
2) A set of services as a proof of concepts, including auto-completion, consistency checking, type inference, and rename refactoring.
3) An evaluation of feasibility and applicability, that is open accessed along with the tool implementation.

II. Motivating Example & Background

This section presents a polyglot code snippet and then discusses background topics required for the rest of the paper.

A. Motivation

Listing 1, shows an example program that uses the GraalVM polyglot API to mix Python and JavaScript code. It consists of Python code, which also executes JavaScript code through the call to `polyglot.eval` at line 18. At lines 12 and 14, we can also see the use of the `polyglot.export` function and its complement `polyglot.import` at lines 19 and 21 in the JavaScript code, enabling data exchange.

As the comments in the code highlight, there are incorrect uses of these functions present in the program. For instance, the variable "Triangle" is imported in the JavaScript code on line 21, but was never exported beforehand. If not taken care

```
import polyglot                                                    1
class Polygon:                                                     2
    def __init__(self, sides=4,                                    3
        name="Square"):                                            4
        self.sides = sides                                         5
        self.name = name                                           6
    def __str__(self):                                             7
        return self.name                                           8
            + f"has {self.sides} sides"                            9
testPolygon = Polygon()                                            10
                                                                   11
polyglot.export_value("Square",testPolygon)                        12
# Pentagon is exported but never imported                          13
polyglot.export_value("Pentagon",                                  14
    Polygon(5,"Pentagon"))                                         15
                                                                   16
# Triangle is imported but never exported                          17
polyglot.eval(language="js", string= "" +                          18
    "var s = Polyglot.import(\"Square\");" +                       19
    "console.log(s.toString());" +                                 20
    "var t = Polyglot.import(\"Triangle\");" +                     21
    "console.log(t.toString());")                                  22
```

Listing 1: Example GraalVM Polyglot Python and JavaScript snippet

of by the programmer, at runtime these errors may lead to null values where an object would be expected or other unhandled exceptions. Unfortunately, unlike for single language programming, current state of the art does not provide design-time support to detect these potential errors in polyglot code. Moreover, the IDE can not give any typing information for the polyglot variables *Square*, *Pentagon*, and *Triangle* that are exchanged, nor can it provide any automated support for refactoring them and their code usages.

To provide any design time tooling for polyglot code, there is first a need to build an AST from it. Once obtained, this AST can be the basis to perform polyglot code analysis, such as variable lifespan analysis, build defined/used pairs, perform type-checking and a variety of other tasks. However, obtaining this AST is not a trivial task as there are multiple qualities a Polyglot AST requires on top of regular AST properties, as well as challenges to ensure these qualities.

B. Background

Polyglot programming is simultaneously a fairly recent concept and a relatively old practice. A lot of runtimes actually implement some form of polyglot interoperability without explicitly naming it. For example, the JVM supports using Java objects in Scala or Kotlin, and there exist libraries that allow using SQL queries in many languages. Recently, more efforts have been put into making more explicit polyglot environments that group languages together.

Examples of solutions are GraalVM[1], LLVM[2], WebAssembly[3]. Existing studies relied on GraalVM to support additional functionalities of polyglot programming [11], [12], [13] and recent studies started to investigate and rely on polyglot programming of GraalVM in various domains [14], [15], [16].

[1]https://www.graalvm.org/
[2]https://llvm.org/
[3]https://webassembly.org/

GraalVM [11] is an Oracle Labs project that aims to create a virtual machine capable of running code from any language at high speeds. Its notable component here is the Truffle language framework, a Java based framework which allows the implementation of any language as an AST interpreter that can then be run by Graal. It currently supports JVM-based languages, LLVM-based languages, Python, JavaScript, Ruby, and R. GraalVM thus provides the polyglot functions `polyglot.eval("language", "code")`, `polyglot.export` and `polyglot.import`. The first will evaluate a piece of code and return its value. The two others allow to expose and retrieve a value of a variable or method to use in a different language than it is defined in.

Our prototype implementation (described in more details in Section IV) bases itself on the previously described GraalVM polyglot runtime environment, but also leverages the Language Server Protocol (LSP) for its analytical needs. LSP is an open source framework promoted by Microsoft that allows a text editor to communicate with a language server that provides code analysis and editing features such as auto completion or type checking. As it is a fairly popular open source framework, it also boasts a variety of language servers implementations featuring most GraalVM supported languages.

The core idea of LSP is that a text editor, also called language client, is not actually required to do any local analysis in order to provide code editing services. Instead, it will send the program's source code to a language server, which will perform the analysis and reply to the client with the results of its analysis, which the language client can then display to the user. This presents the advantage of being editor-agnostic as any IDE can implement a language client interface and reuse existing language servers instead of implementing its own static analysis system. It also potentially saves local computing power if the language server is on a distant machine. More information is available on the official LSP website[4].

III. Polyglot AST approach

This section presents our envisioned approach for a Polyglot AST. First, it gives an overview of how we construct a polygot AST. Then, it discusses the requirements that the polygot AST must satisfy. Finally, it addresses the existing challenges and illustrates solutions.

Figure 1 depicts the workflow of our approach to construct a Polyglot AST. Given polyglot code, we first parse the various code written in different languages (step 1). We then unify the different ASTs by first identifying the AST nodes that represent the polyglot constructs to communicate with other languages (step 2). These nodes are initially parsed as calls to unknown methods, which we can transform into explicit AST nodes of polyglot method calls. This allows for a tree traversal of the Polyglot AST while being aware of the different languages used in each subtree. Finally, services of code analysis can be built on top of the Polyglot AST, such as a rename refactoring or a consistency mechanism (step 3).

[4]https://microsoft.github.io/language-server-protocol/

A. Overview of a Polyglot AST

This section discusses the three needed generic properties that are expected of a Polyglot AST and are necessary in order to use a Polyglot AST in the same way as a regular AST. These properties are the key factors that allow tool developers to later build services on top of the AST.

1) AST shape homogeneity: The first property is shape homogeneity of nodes, i.e., uniformization of the AST structure. This means that regardless of their type, metadata or other attached information, all nodes of the tree should respond to the same primitives in a similar manner. While this is usually guaranteed and trivial for regular, single language ASTs, it may not be for polyglot ASTs. At the very least, a Polyglot AST shall be agnostic of the language of the node it is currently treating. That is, the language associated with the node should not change the manner in which to query the next node to process. In particular, the language associated with the node should not change the manner in which to query the next node to process. For example, if a function `nextStat()` can return the next statement in one language, it should behave the same in the rest of the languages without requiring the service to be aware of the language to know which function to call.

2) Identification of polyglot constructs: The second important property is the clear identification of polyglot-specific constructs in the program. In polyglot code, there are explicit constructs that allow languages to interact and mix together. Those constructs need to be reflected in the AST, in a manner distinguishable from language-specific constructs. At the very least, there needs to be a way for a service to be aware of a language change in a polyglot program. Other polyglot constructs also need to be clearly marked in the AST, e.g., using special nodes or edge types (in Figure 1 this is represented by the red nodes). These markers allow services to easily implement polyglot specific features and as such need to be available directly in the AST, rather than having the service do a more complex check to spot polyglot constructs.

3) Design time AST: Finally, the last set of properties is related to the nature of the services. As the AST is intended to provide a basis for design time analysis, it needs to support real-time editing of the code base. This entails two main characteristics, namely the support of incremental parsing and resilience against syntax errors.

We assert that it is possible to use a polyglot AST that meets the three identified requirements as a foundation for a variety of services that can support design time program analysis. There are a number of problems to tackle in order to be able to build a Polyglot AST and meet the above requirements. Let us illustrate by running through the steps in our approach showcased in Figure 1.

B. Polyglot AST construction: parsing

The first step to any kind of analysis is parsing of the source code. This is a research field in and of itself, but the task has become fairly straightforward in most cases due to the availability of parsing tools adapted for various languages.

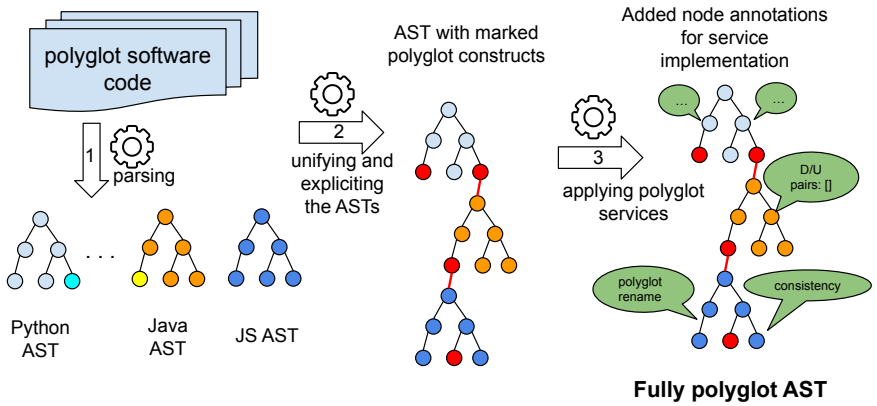

Fig. 1. Workflow of a conversion from polyglot code to Polyglot AST.

However, in our case this is not as simple of a task. First, while there exists different tools and frameworks that are able to parse a program's source code, they are geared towards operating on a single language's syntax. As such, in a polyglot programming setting, they will be unable to produce an exploitable AST for the purpose of polyglot analysis.

There are two scenarios where a single-language parser will be unable to recognize a polyglot programming construct. Depending on how the polyglot runtime is defined, the embedding of a language within another can be done either using a new syntactic marker that was not already present in the base language (a "polyglot operator") that would signal a language switch, or using preexisting constructs within the base language such as function calls.

In the former, the solution is simple albeit not effortless: the parser grammar rules simply need to be updated in order to recognize and support this operator. In the latter however, there are more options; it is still possible to overhaul the parser grammar rules to recognize the polyglot usage case of the language construct, but another choice is to let the language parser build its AST normally, and modify it afterwards so that the nodes corresponding to the polyglot constructs actually have a special type and behavior.

In both cases however, simply recognizing the polyglot language switch is not sufficient as we still have not parsed the code it contains. Ultimately, which solution is picked is an implementation choice that will depend on the exact framework or tool being used. The first option is to modify the parser by adding rules that handle the syntax of all languages that may be used in a polyglot setting. However, while optimal in regards to performance, this requires extensive work - including the complete re-implementation of a parser for each of the supported languages.

A more concise and easily extensible method is to reuse existing parsing tools dedicated to specific languages. As most languages there will already have preexisting extensive work geared towards providing an AST for a program, we can simply reuse these tools to parse polyglot programs containing these languages. This saves implementation effort, and lets us simply fetch a corresponding parsing tool if we ever need to add support for a new language to our system. Then, we have a set of ASTs that are fully parsed and ready for the next step.

C. Polyglot AST construction: unification and marking

Parsing the source code is necessary towards building a polyglot AST, but we have yet to guarantee the three properties mentioned in our overview. In this sub-section, we will cover the first two properties. As the third property is more closely linked to implementation rather than methodology, we will only discuss it briefly in the next subsection.

One of the downsides of choosing to reuse existing parsing tools specialized for single languages is that those tools were not thought out to interoperate with each other. This means that natively, the system will not preserve shape homogeneity if we simply merge the output ASTs together.

As a reminder, shape homogeneity means that the interface for traversing and retrieving node information from the AST should be identical regardless of which single-language sub-AST a user is manipulating. In other words, if we are working with multiple parsing tools to handle various languages, we need to modify the functions used to retrieve node information or select nodes. The most straightforward way to realize these modifications is to use a wrapper that will manage requests and forward them to the underlying AST.

Once the AST format is uniformized to the same interface across all sub-trees, we can take a look at the second property, identification of polyglot constructs. This property has two different aspects to consider: language switch constructs, and secondary polyglot constructs.

Language switch construct refers to the syntactic or semantic construct that enables the actual embedding of a second language within a program. As mentioned previously, this construct can take many forms such as an operator, file metadata, a specific function name or any different implementation, and depends on the runtime environment being used. Nonetheless, in our polyglot AST system, an occurence of a language switch should always translate to one AST node whose first child is the root of the embedded code's sub-AST of the target foreign language. This sub-AST should also be the node defined as the next one in the case of a depth-first traversal of the tree.

Secondary polyglot constructs refers to any other polyglot interaction tools provided by the runtime environment that are

not language switches, such as polyglot bindings import and export functions in GraalVM[5]. As the definition of secondary constructs is highly dependent on which polyglot runtime environment is being used, the nodes representing these constructs may take any form so long as they are unambiguously recognizable as polyglot nodes, and are able to provide all of the relevant information.

The marking of both of these constructs as specific types of nodes can happen either during the parsing phase, or as a post-processing of the multiple ASTs. In both cases however, to preserve shape homogeneity, the polyglot nodes should be identical in format regardless of the language they were parsed in. This does assume that in the polyglot environment being used, a polyglot construct provides the same information in all languages in the source code. Once these constructs are specialized into polyglot nodes and the set of ASTs is linked by all of the language switch nodes, it is possible to exploit the AST in order to apply analysis services.

D. Polyglot AST construction: application of services

With a polyglot AST that fulfills the shape homogeneity and polyglot construct identification properties, we can provide polyglot services useful to developers.

By leveraging shape homogeneity, we ensure that it is possible to traverse the polyglot AST in the same manner a single language AST could be traversed, enabling the same analysis techniques to be reproduced across a polyglot program. By explicitly marking polyglot constructs as special nodes within the AST, we also enable analysis services specifically geared towards polyglot programs. By combining both of these factors, we are thus able to treat a polyglot program as a single polyglot AST, and apply services that relate program points without being limited by language boundaries.

The third property relates mainly to flexibility and performance aspects. Because services may be useful to a developer currently editing the code base, the AST needs to be able to work with source code that is undergoing changes. This means that syntax errors for instance should not crash the AST builder, and still provide a reasonably exploitable AST. A second important point is that for many services, the developer should not have to wait exceedingly long before an error is raised. This means the AST should be able to both parse the source code and apply services within acceptable delays, probably using incremental parsing and various optimization techniques.

IV. Prototype Implementation

This section details a prototype example implementation based on our approach. We will first detail the frameworks and give an intuition of how our implementation functions, and then present four example services that illustrate how a polyglot AST can be used to help developers.

Our prototype implementation is made publicly available through GitHub. It is split in two major parts: the framework we call Polyglot AST and the Polyglot Language Server extension for Visual Studio Code. The first is a proof-of-concept framework showcasing our approach described in section III and providing a Polyglot AST. The second is an extension for the IDE Visual Studio Code, basing itself on the Polyglot AST to offer the aforementioned four services.

[5]https://www.graalvm.org/22.2/reference-manual/polyglot-programming/

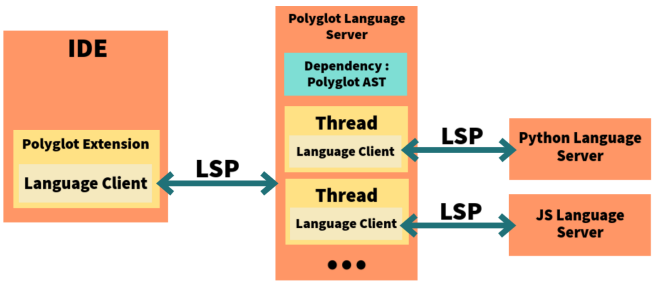

Fig. 2. Polyglot language server structure with LSP.

A. Polyglot AST

The aptly named Polyglot AST framework[6] is a proof-of-concept implementation of a polyglot AST builder for the GraalVM runtime environment. It handles parsing and building of ASTs for programs in a mixture of Python and JavaScript so far, and is designed to let a multitude of polyglot code analysis services to be implemented.

For its parsing needs, Polyglot AST uses tree-sitter[7], a parsing framework that supports many languages. Tree-sitter also supports incremental parsing and error nodes in case of syntax errors in the source code. Because it is a single framework supporting many languages, it uses the same node structure for all of them. These qualities greatly facilitate ensuring both the shape homogeneity and the design time AST properties, meaning we will mostly have to worry about the unification of the ASTs and the marking of polyglot constructs in our implementation.

As we did not modify tree-sitter's grammar rules, every time we retrieve an AST for a program, we have to traverse it and retrieve polyglot constructs. We can then both modify the corresponding nodes to have a specific polyglot type, and also register language switches which then lets us parse the embedded polyglot programs and repeat this process until we get the full polyglot AST.

Polyglot AST is implemented as a simple wrapping system written in Java, where we implement the links between sub-ASTs of the program through a hash-map of the language switch nodes to the sub-ASTs. We also provide a simple interface to implement services, that lets a user traverse the tree and access node type to implement operations. This interface is leveraged by the Polyglot Language Server to implement a variety of services.

B. Polyglot Language Server

The Polyglot Language Server[8] is a language server implementation provided with a Visual Studio Code extension for polyglot programs written in GraalVM, and bases itself on the Polyglot AST framework. The way it functions is summed up in Figure 2 and in the following.

On the IDE side, the actual extension runs a regular language client, simply sending over the polyglot source code to the language server process. The language server then builds the Polyglot AST, and uses it to apply services.

These services also leverage the fact that there exist language servers dedicated to the programming languages present in the source code. This saves us the trouble of re-implementing potentially complex, pre-existing analysis algorithms for the languages and lets us focus on the polyglot logic. We can thus implement the following four polyglot services.

1) Service 1: Consistency diagnostics: Consistency diagnostics are a set of sanity checks done on a polyglot program. They mostly consist in finding usages of polyglot construct that look like they might be programming mistakes such as unused polyglot variables or polyglot evaluation of files that do not exist. Those usages are then signaled to the programmer so they can potentially correct their code. A list of diagnostics currently supported is available in table I.

This is currently implemented by traversing the AST after every change and recording inconsistencies, ignoring most control flow instructions. As such, the service might raise warnings on a correctly written program. This could be with additional language specific consistency checking.

2) Service 2: Type checking and inference: The Polyglot Language Server includes a best-effort type checker, which on hover over a polyglot variable, attempts providing a type. This can be of great help in some cases where exchanges between languages do not always make it clear which type a variable might have. Figure 3 summarizes the principle of this diagnostic.

3) Service 3: Auto-completion: Auto-completion of polyglot variables lets a user request their IDE to provide a list of currently available polyglot variables they can use, and generates the necessary code to import them locally. This works by listing all exported polyglot variables, filtering out the ones that have already been imported to the current file, and presenting the available completion options for the user to select. This helps avoid accidentally using undefined variables and saves a bit of time. It also does not require a developer to double check their syntax.

4) Service 4: Renaming: Renaming of polyglot imports and exports lets a user opt to rename all occurrences of a variable in their source code, which avoids having to skim through source code whenever making a change. It spots all occurrences of a specific variable in the code base, and automatically modifies all of them together when triggered, which can be done through a right click.

[6]https://anonymous.4open.science/r/PolyglotAST-D67C/README.md

[7]https://tree-sitter.github.io/tree-sitter/

[8]https://anonymous.4open.science/r/polyglot-language-server-CB46/README.md

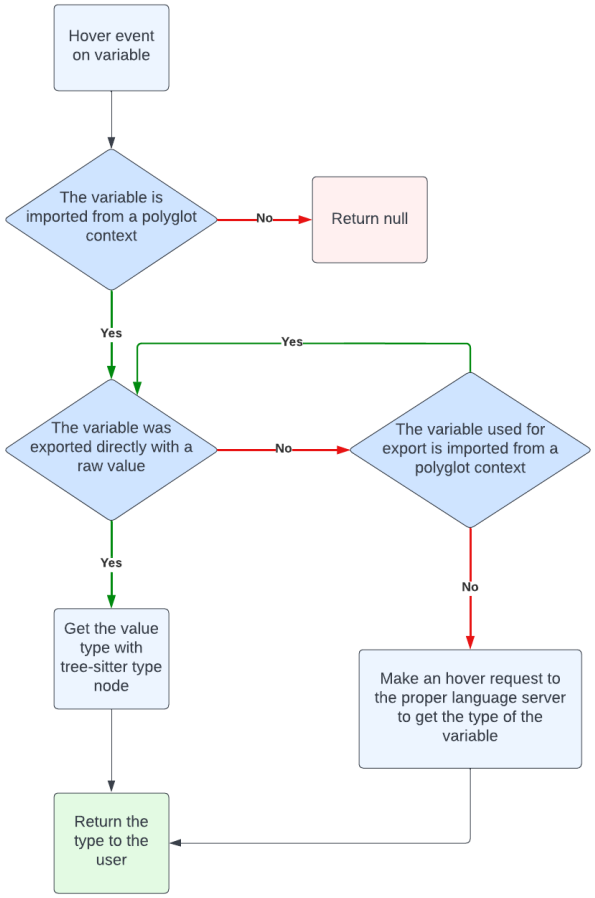

Fig. 3. Type checking algorithm.

V. Evaluation

This section evaluates our Polyglot AST and its services. In particular, we assess its feasibility and applicability. First, we present the research questions, the data set, and the evaluation protocol. Then we detail the results. We finally discuss the threats to validity, and the scope of the approach. We ran the implementation of our approach on the following hardware configuration: *4 x AMD® Ryzen 5 2500u CPU @ 2.0-3.6GHz; 8GB ram; 500 GB SSD*, running *Ubuntu 22.04.1 LTS*.

A. Research Questions

We formulate the following research questions to assess feasibility and applicability, namely:

RQ1 *Can we build at scale a Polyglot AST on software projects using GraalVM polyglot features ?* This aims to investigate the feasibility of our approach to correctly provide a Polyglot AST and further assess its performance.

RQ2 *What are the performances of our services on the built Polyglot AST ?* This aims to investigate the applicability of the services of our approach and their performance in practice. In particular, we answer the sub-questions below:

TABLE I
DIAGNOSTICS SUMMARY

Diagnostic	Description	Type
Evaluation File not found	Occurs if you wrote a polyglot evaluation of a file that doesn't exist	Error
Unused Export	Occurs if you exported a variable but never imported it back.	Info
Import Before Export	Occurs if you import a variable that is exported after the import statement	Warning
Import without Export	Occurs if you import a variable that has never been exported in a polyglot context	Warning
Useless Variable Import	Occurs if you import a variable that was exported previously from the same file	Info
Same File Evaluation	Occurs if you write a polyglot evaluation of the file you are currently in	Warning

TABLE II
DATA SET DESCRIPTION.

Projects	1st Polyglot Program	2nd Polyglot Program	3rd Polyglot Program
Python	1.6k LOC	16k LOC	17k LOC
JavaScript	1.4k LOC	10k LOC	80k LOC
Total	3k LOC	26k LOC	97k LOC

RQ2.1 What are the performances of the polyglot consistency checking service ?
RQ2.2 What are the performances of the polyglot auto-completion service ?
RQ2.3 What are the performances of the polyglot rename refactoring service ?
RQ2.4 What are the performances of the polyglot type inference service ?

B. Data Set

The source of our data set is GitHub. We aimed to select Python and JavaScript projects that we can first compile and run. As polyglot techniques are still emerging, few open source projects are available to evaluate our framework. To circumvent this issue, we selected existing non-polyglot JavaScript and Python code and then introduced GraalVM polyglot functions, with both correct and incorrect usages to evaluate our Service 1. As we mainly aimed to evaluate time performance, we did not need to adapt these usages to fit the semantics of the projects. For example, by implementing a Python function in JavaScript and replacing its original call in Python by the new equivalent one in JavaScript.

In the end, we combined [P1][9] with [P2][10], [P3][11] with [P4][12], and [P5][13] with [P6][14]. Table II details the selected projects and their combinations. Thus, we have small, medium, and medium-large sized projects.

[9]https://github.com/bendc/sprint/blob/master/sprint.js
[10]https://github.com/spadgos/sublime-jsdocs/blob/master/jsdocs.py
[11]https://github.com/Jollify-Software/Jollify-Software.github.io/blob/main/demos/juel.js
[12]https://github.com/HuyaneMatsu/hata/blob/master/hata/discord/client/client.py
[13]https://github.com/bbqmq/obnovaa/blob/main/atom%20(1).py
[14]https://github.com/shaileshjanovis/shopify/blob/main/assets/index.js

C. Experimental Protocol

We then ran our tool on these projects and measured the time to build the Polyglot AST as well as the time needed for the polyglot consistency checking, auto-completion, rename refactoring, and type inference.

For all projects, we varied the number of imports and exports between 50, 100, and 150, while also varying validity of usage to test multiple scenarios. In each case, we introduced one GraalVM `Polyglot.Eval()` call to link files together, and then valid or invalid exports and import calls before and after the eval call. We ran the different measurements of the Polyglot AST creation and the services 100 times, and we report on the average execution time.

D. Results

We now present and discuss the observed results.

1) RQ1: To answer RQ1, we built the polyglot AST for the three projects with 10, 100, and 500 imports and exports, as shown in Figure 4. We observe that the construction time does not vary and is not dependent on the number of polyglot links of imports and exports, but rather on the code size. It varies from 34 milliseconds for the small project to around 1097 milliseconds on average for the largest project. In the largest project, it varied respectively from 1097, 1059, and 1084 milliseconds on average with 50, 100, and 150 imports and exports.

Moreover, we also compared the overhead of computing the Polyglot AST compared to simply parsing the different code files (in our case the two Python and JavaScript files). Figure 5 shows that there is no observed overhead of computing the Polyglot AST, with almost equivalent time of parsing separately the different target code and the polyglot AST.

2) RQ2: We now answer the questions related to the performance of our polyglot services. In the following, we ran 100 times a diagnostic of consistency checking, 100 random rename refactorings, 100 random auto-completion, and 250 random type inference. In the above Figures, we report on the average time that we measured in the several runs.

a) RQ2.1: Regarding the consistency checking service, we observe marginal execution time from 13 milliseconds up to 467 milliseconds on average. We also observe that this service does not depend on the number of polyglot links of imports and exports. In the biggest project, it varied respectively from 355, 396, and 467 milliseconds on average with

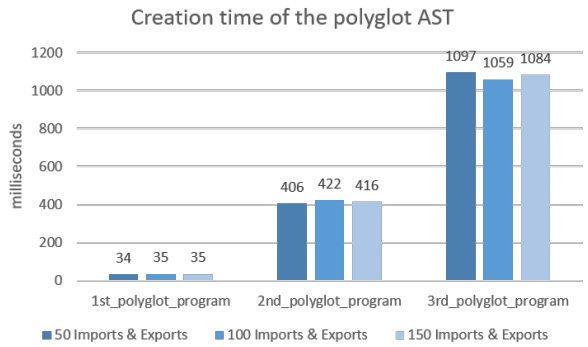

Fig. 4. Performance of creating the polyglot AST.

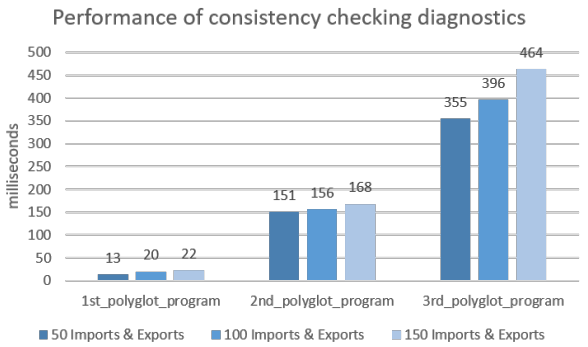

Fig. 6. Performance of the consistency checking polyglot service.

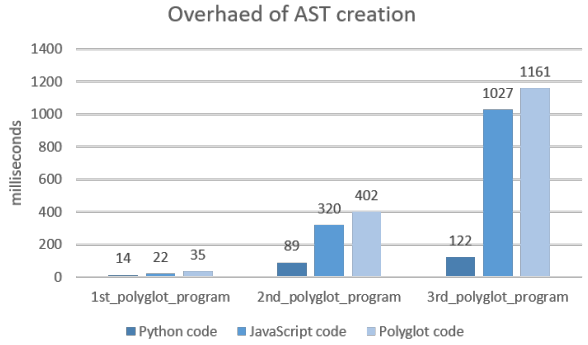

Fig. 5. Measurement of the overhead for the creation of the polyglot AST.

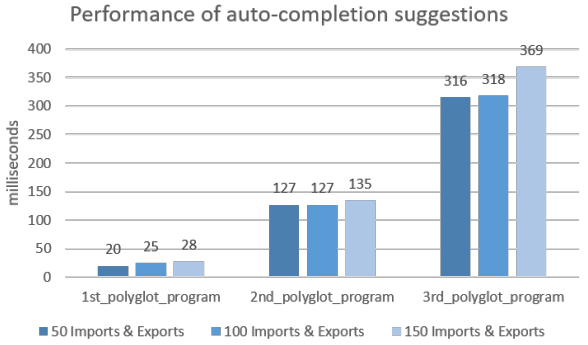

Fig. 7. Performance of the auto-completion polyglot service.

50, 100, and 150 imports and exports including inconsistencies from Table I.

b) RQ2.2: Regarding the performance of the auto-completion. We observed a similar trend as with the consistency checking service, costing from 20 up to 369 milliseconds on average of computation regardless of the number of imports and exports. In the biggest project, it varied respectively from 316, 318, and 369 milliseconds on average with 50, 100, and 150 imports and exports.

c) RQ2.3: Regarding the performance of the rename refactoring. We similarly observe marginal execution time from a few milliseconds up to 740 milliseconds on average. In the largest project, it varied respectively from 705, 673, and 740 milliseconds on average with 50, 100, and 150 imports and exports.

d) RQ2.4: Finally, regarding the performance of the type inference. We observed that the performances depends on the way we infer the type, either from the polyglot AST or from the LSP of the target language. We observe that the performance does not depend on the number of polyglot imports and exports. Rather it is stable and depends on how the types are inferred. It varied from 156 to 171 milliseconds on average when using the polyglot AST and varied from 331 to 344 milliseconds on average when redirecting a request to the target language LSP.

E. Discussion and limitations

Overall, we observed that we were able to compute the polyglot AST accurately by unifying the various ASTs of the target languages. More importantly, with no overheard when constructing the polyglot AST while also providing scalable polyglot services withing less than a second for an almost 100k polyglot project.

Our goal with the current approach is to provide a basis for building design time support for polyglot programming in an IDE. Thus, other services can be implemented as well on top of the polyglot AST. For example, handling more types of refactorings, etc.

However, our current solutions is limited to only supporting Python and JavaScript polyglot code. Integrating other languages is left for future work. Beyond the supported languages, our implementation is also limited to GraalVM polyglot programs. Yet, this is only an implementation limitation, as conceptually, the polyglot AST is made to be generic. Our implementation also only supports subprograms passed as direct static string input and not through variables, which limits programming styles.

F. Threats to validity

This section discusses threats to the internal, external and conclusion validity of the results presented above.

1) Internal Validity: In our evaluation, the internal threats to validity are centered on the way we make the projects polyglot with the imports and exports polyglot links. In fact, we first make one program call another one with the GraalVM *polyglot.eval('file')* call function, before to randomly inject the imports and exports in both files. We made sure that the syntactically the injection is correct. However, we also covered both correct and incorrect usages. For example, an injection of both an export of a variable *var* and its import, which is

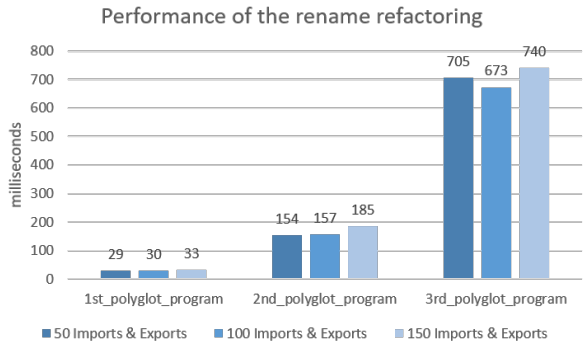

Fig. 8. Performance of the rename refactoring polyglot service.

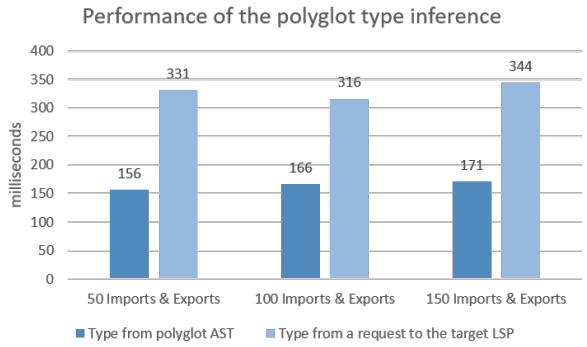

Fig. 9. Performance of the type inference polyglot service.

a correct usage. Whereas, we also inject an import without an export and vice-versa to be detected by our consistency checking diagnostic. Similarly, for the rest of the services, we randomly applied the auto-completion, rename refactoring, and type inference, while varying the number of imports and exports to observe what would be the effect on performance. Finally, we repeated the measurements 100 times and we reported on the average for the polyglot AST creation and the different implemented services.

2) External Validity: In this paper, we evaluated only on polyglot projects consisting of Python and JavaScript, while relying on GraalVM. Tree-sitter is able to handle many more languages, but adding them to our framework is left to future work. Thus, in our current evaluation we cannot generalize our results for other polyglot projects involving other languages. In particular, more than two languages, such as involving at the same time Java, Python, JavaScript, Ruby, R, C, C#, C++, etc. Further experimentation is necessary herein in future work.

3) Conclusion Validity: Our evaluation shows the feasibility and applicability of building a polyglot AST ad using to provide polyglot services, namely auto-completion, consistency checking, type inference, and rename refactoring. However, we only evaluated on three polyglot projects of Python and JavaScript code from 3k up to 100k LOC in the largest project. Thus, we cannot conclude on the observed performance for any polyglot program. To have more statistical evidence, we plan to evaluate on more polyglot project involving more than two languages other than Python and JavaScript.

VI. Related work

This section discusses related works to the topic of polyglot code analysis as well as a few polyglot execution platforms. *Polynote*[15] is a polyglot notebook that supports Scala, Python, and SQL, where each cell can use a different language. Cells also share variables in a common scope. As with many notebooks such as Jupyter, there is the possibility of adding either text or code, as well as display output from running individual cells. Interoperability in Polynote appears quite trivial to use, as it follows the principle of notebooks where all cells execute within the same scope. Thus, any variable defined in a cell is also available in the following cells, even if they are not ran using the same language. This way, multiple languages can interact freely as they share a common variable scope. However, it is limited to only sharing data across languages that is of a primitive type. Anything more complex than an array is not able to be passed from a language to another.

Eco [9] is a language composition editor that also acts as a runtime environment for Python, SQL and HTML code. It is a prototype IDE that allows mixing languages in a polyglot manner through ”language boxes”, a way of embedding a language within another similar to Graal’s code evaluation.

Eco is mostly intended as a prototype polyglot IDE and thus does not support many languages or types of analysis. However, this approach could easily be generalized to implement more runtimes or services and thus allow practical usage. The approach differs from ours in the sense that Eco uses a all-in-one system where the IDE constitutes both the runtime environment as well as the analysis system, rather than explicitly separating them as different entities which can be used in a modular way.

Kreindl *et al.*[17] present an adaptation to a polyglot environment of Dynamic Taint Analysis, a technique which can be used to detect many software vulnerabilities. They present a platform designed to work with GraalVM and Truffle and provide polyglot taint analysis, which uses both language specific and language agnostic taint propagation rules in order to perform a dynamic analysis.

Niephaus *et al.*[18] propose a method to build language-agnostic runtime tools, based on LSP and GraalVM. They illustrate their approach through an Example-based Live Programming system and provide insight on how to extend their system, as well as ideas on how to build more language-agnostic tools that rely on run-time information.

Both of these previous work provide great insight on what can be done with dynamic and run-time analysis of polyglot code, whereas we attempt to provide more insight on static and design-time analysis. These techniques are complementary in current program analysis techniques, and we believe they also are in a polyglot environment.

Riese *et al.*[19] focus on making polyglot language interaction easier by allowing developers to define the interface they expect polyglot messages to match. This lets reuse of

[15]https://polynote.org/

polyglot frameworks and libraries be done in a more high-level manner, based on their exact needs for their application, and lowers the technical charge of having to learn the exact polyglot environment's interface.

On a similar theme of facilitating reuse of polyglot code, Ehmueller *et al.*[10] proposes an approach to finding libraries and projects which best fit developers needs. By adapting current approaches to enable the searching of code written in different languages, the process of finding a suitable project to reuse can be greatly shortened and automated to some degree.

The example services provided in section IV complement the benefits of these works, by helping developers get information on the polyglot libraries they may reuse through typing of imported variables or auto-completion.

Many works in the literature focus on polyglot programming as a runtime environment. As such, they primarily tackle issues relating to execution such as performance [20], [11], [21] or ease of adding support for new languages[22]. A reader interested in discovering more topics related to the implementation of high performance polyglot run-times might take a look at the GraalVM Github Publications[16] section.

Our work is in the continuity of studies such as Peterson[15] that examine the impact of polyglot programming on developers, as our aim is to enable analysis of polyglot code at design time for developers in order to improve development efficiency. To the best of our knowledge, while there are many works who provide examples of run-time analysis of polyglot programs, there were no previous approaches that provided a generic method to ease polyglot code analysis at design time.

VII. Conclusion and Perspectives

In this paper, we presented an approach that enables the global analysis of polyglot code by building a uniform and polyglot AST of it. We motivated this challenge and identified the main requirements for building a polyglot AST. We implemented a proof of concept prototype of our solution on GraalVM's polyglot API, in a VSCode extension with an LSP-based implementation. On top of the polyglot AST, we demonstrated the ability to implement several polyglot-specific analysis services, namely auto-completion, consistency checking, type inference, and rename refactoring. Evaluation on three polyglot projects of 3k, 26k and 97k LOC involving Python and JavaScript code showed that we could build a polyglot AST supporting update without significant overhead. The various automated polyglot-specific analyses showed to scale well with few milliseconds and up to less than a second of execution in the largest polyglot project. As a part of our future work, we plan to to extend the approach to more languages other than Python and JavaScript, such as Java, C#, C, C++, R, and Ruby. We also plan to implement other polyglot services and empirically evaluate the Polyglot AST on other projects with more than two involved languages.

[16]https://github.com/oracle/graal/blob/master/docs/Publications.md

References

[1] E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, and A. Garcia, "Code and commit metrics of developer productivity: a study on team leaders perceptions," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2519–2549, 2020.

[2] A. Mockus, "Succession: Measuring transfer of code and developer productivity," in *2009 IEEE 31st ICSE*, pp. 67–77, IEEE, 2009.

[3] O. Dieste, A. M. Aranda, F. Uyaguari, B. Turhan, A. Tosun, D. Fucci, M. Oivo, and N. Juristo, "Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study," *Empirical Soft. Engineering*, vol. 22, no. 5, pp. 2457–2542, 2017.

[4] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing developer assistant: improving developer productivity by recommending sample code," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on FSE*, pp. 956–961, 2016.

[5] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. on soft. eng. TSE*, vol. 36, no. 4, pp. 546–558, 2009.

[6] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Improving code readability models with textual features," in *2016 IEEE 24th ICPC*, pp. 1–10, IEEE, 2016.

[7] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, p. e1958, 2018.

[8] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *2019 IEEE/ACM 27th ICPC*, pp. 2–12, IEEE, 2019.

[9] L. Diekmann and L. Tratt, "Eco: A language composition editor," in *SLE*, pp. 82–101, Springer, Sept. 2014.

[10] J. Ehmueller, A. Riese, H. Tjabben, F. Niephaus, and R. Hirschfeld, "Polyglot code finder," in *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, ¡Programming¿ '20, (New York, NY, USA), p. 106–112, Association for Computing Machinery, 2020.

[11] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proc. of Onwards!'13*, pp. 187–204, 2013.

[12] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "Graalsqueak: toward a smalltalk-based tooling platform for polyglot programming," in *Proc. of MPLR'19*, pp. 14–26, 2019.

[13] F. Niephaus, T. Felgentreff, and R. Hirschfeld, "Towards polyglot adapters for the graalvm," in *Onwards!'19*, 2019.

[14] J. A. Romero-Ventura, U. Juárez-Martínez, and A. Centeno-Téllez, "Polyglot programming with graalvm applied to bioinformatics for dna sequence analysis," in *Proc. of CIMPS'21*, pp. 163–173, Springer, 2021.

[15] C. S. Peterson, "Investigating the effect of polyglot programming on developers," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–2, IEEE, 2021.

[16] A. Parravicini, A. Delamare, M. Arnaboldi, and M. D. Santambrogio, "Dag-based scheduling with resource sharing for multi-task applications in a polyglot gpu runtime," in *Proc. of IPDPS'21*, pp. 111–120, 2021.

[17] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck, "Multi-language dynamic taint analysis in a polyglot virtual machine," in *17th International Conference on Managed Programming Languages and Runtimes*, MPLR 2020, p. 15–29, 2020.

[18] F. Niephaus, P. Rein, J. Edding, J. Hering, B. König, K. Opahle, N. Scordialo, and R. Hirschfeld, "Example-based live programming for everyone: Building language-agnostic tools for live programming with lsp and graalvm," Onward! 2020, p. 1–17, 2020.

[19] A. Riese, F. Niephaus, T. Felgentreff, and R. Hirschfeld, "User-defined interface mappings for the graalvm," in *4th International Conf. on Art, Science, and Engineering of Programming*, ¡Programming¿ '20, p. 19–22, 2020.

[20] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters," *SIGPLAN Not.*, vol. 48, p. 73–82, oct 2012.

[21] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime," *SIGPLAN Not.*, vol. 51, p. 78–90, oct 2015.

[22] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger, "A domain-specific language for building self-optimizing ast interpreters," in *GPCE: Concepts and Experiences*, GPCE 2014, p. 123–132, 2014.