Language-Independent Development Environment Support for Dynamic Runtimes

Daniel Stolpe
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
daniel.stolpe@student.hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

Christian Humer
Oracle Labs
Zurich, Switzerland
christian.humer@oracle.com

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

There are many factors for the success of a new programming language implementation. Existing language implementation frameworks such as Truffle or RPython have focused on run-time performance and security, or on providing a comprehensive set of libraries.

The tools that language users need to be productive are also an important factor for the adoption of a programming language. The aforementioned frameworks, however, provide limited support for creating comprehensive development tools. This is an impediment to language adoption that can be as serious as performance issues or lack of libraries. Both Truffle and RPython already provide run-time tools such as for debugging, profiling, or coverage in a language-independent manner, but neither support static development tasks carried out in code editors.

In this work, we propose a language-agnostic approach to add this missing tooling by making the Language Server Protocol available automatically to all language implementations on the Truffle framework. Furthermore, we show how our approach can utilize runtime information to provide IDE features rarely available for dynamic programming languages.

CCS Concepts • **Software and its engineering** → **Integrated and visual development environments**; **Software maintenance tools**.

Keywords Language Server Protocol, Truffle, Graal

ACM Reference Format:
Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-Independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3359619.3359746

1 Introduction

As use-cases evolve, so do programming languages to deal with the rising complexity of modern software systems. Developing a new programming language from scratch is a challenging task. Traditionally, ensuring the performance of new implementations is competitive with existing ones has consumed a large amount of development time. Frameworks such as RPython [1] and Truffle [21] address this and provide good performance with reasonable effort on the part of the language developer.

However, another barrier for new and evolving languages has been tool support. Users often have to give up on existing tools and learn new ones, if there even are replacements for all the tools they used before. The Truffle and RPython frameworks have recognized this and thus provide hooks for language implementations to gain support for interactive debugging, profiling, and coverage information easily.

One tool that is still missing is a code editor with a rich set of capabilities. The Language Server Protocol (LSP) tackles this tooling aspect by connecting multiple Development Environments (DEs) with a *language server*. This server is implemented per language, but the burden of building or integrating with existing environments is handled by the protocol.

In this work, we propose an approach to provide assisted code editing for new language implementations in a framework like Truffle with the same ease as it provides performance. Language implementers merely have to use the hooks provided by the framework in order to get an integration with any LSP client for free, and receive at least basic syntax error reports, completion of globals and locals, as well references, definition, and meta-information lookups. Our language-agnostic approach is not meant to replace established language servers for a specific language, but provide a good baseline for languages for which no such server exists yet.

Code assistance for dynamic language code is especially challenging and our approach includes suggestions on how to collect and utilize runtime information to improve editing features for dynamic languages. For programming languages with typed symbols, like Java, the required information can be derived from the source code, but for dynamically typed languages, DEs typically try to infer types and/or make use of additional type hints in source code comments or other annotations. Good type inference engines are difficult to build and require language-specific knowledge. Many dynamic languages also include constructs that prevent type inference such as `eval`.

As proof of concept, we have implemented Graal-lsp, a polyglot language server implementation for the GraalVM. Thus, our contributions in this work are:

- An approach to build a language-agnostic language server on top of performance-focused language implementation frameworks.
- An approach to provide IDE-like features for dynamic languages, by collecting and utilizing runtime information.
- An implementation of these approaches using the Truffle framework, which includes polyglot code completion and goto-definition features.

The rest of this paper is structured as follows: in section 2, we provide background information pertaining to the LSP and language implementation frameworks. In section 3 we describe our approach to build a language-agnostic language server on top of Truffle and what the language developer has to do in order to support the different features. Finally, in section 4 we discuss related work and we conclude and summarize future work in section 5.

2 Context

In this section we give a brief overview over the two areas we combine in this work: a) Truffle and RPython as instances of performance-focused language implementation frameworks, and b) the LSP and how we approached supporting it in a language-independent manner.

2.1 Tools in Language Implementation Frameworks

In this work we are interested in performance-focused language implementation frameworks like Truffle [21] or RPython [1]. In these frameworks, tooling has been a focus only where it did not compete with performance. Thus, languages developed in these frameworks have fewer tools available when compared to language workbenches like Spoofax [7], Xtext [3], or MPS [20], which are used to create Domain Specific Languages (DSLs) and their corresponding DEs.

Truffle [21] is a language implementation framework written in Java. The language developer uses the framework to write an Abstract Syntax Tree (AST) interpreter, which the framework optimizes during interpretation. How source code is parsed and the AST is structured is the responsibility of the language implementation. Truffle has built-in support for developing language-agnostic tools by providing an instrumentation API [16] using meta-information that language implementations attach to their AST nodes. This API has been used already to provide all Truffle languages with code coverage, CPU and memory profiling tools, as well as debugging with the Chrome debugger. (The debugger protocol is also supported by VSCode, IntelliJ, and Netbeans.)

PyPy [13] is an alternative implementation of the Python language [17] written in RPython, a restricted subset of Python. The language developer is free to structure the interpreter as they see fit, although bytecode interpreters are the most common approach for RPython-based languages. RPython provides no generic API for tools like Truffle does. However, the RPython library includes a CPU profiler[1] and a reverse debugger[2]. Languages can make use of these with minimal effort by registering entry points with Python decorators and by implementing callbacks to attach for the tools to call into.

2.2 Code Editors and the Language Server Protocol

A huge number of code editors exist and the choice of editor is an amusingly frequent source of arguments amongst developers. The LSP[3] is an approach to standardize language support for development tools to make the choice of editor less tied to the choice of language. The LSP defines a *server* and a *client*. The server needs to be implemented only once for every language, providing the language-specific development features. Any editor that can act as a language client can make use of any language server and therefore are able to support multiple languages with minimal effort.

For a number of languages (including R, Python, Ruby, and JavaScript), multiple LSP server implementations already

[1]https://morepypy.blogspot.com/2016/08/pypy-tooling-upgrade-jitviewer-and.html, accessed 2018-12-17
[2]https://morepypy.blogspot.com/2016/07/reverse-debugging-for-python.html, accessed 2018-12-17
[3]https://microsoft.github.io/language-server-protocol (accessed 2018-12-17)

exist and these servers can be also be used with the Truffle language implementations. However, both Truffle and RPython already try to obsolete the need for multiple implementations of a GC, JIT, dynamic object model, debugger, profiler, and more. We argue that there should not be a need for multiple implementations of language servers, either.

In this work we focus on how Truffle can (in a language-independent manner) support a subset of the features of the LSP: **1)** navigational features to follow references of names and go to the implementations of functions; **2)** informational features to display syntax errors, documentation, function signatures, and types; **3)** code completion both of names available in local and global scopes as well as properties on names or results of operations; and **4)** code completion when multiple languages are mixed as an example of a feature that other language servers cannot easily support.

3 Approach and Implementation

In this section we will address, in the order given in the previous section, the LSP features we have implemented for the Truffle framework[4]. To support all the LSP features we describe, language implementations on Truffle need to implement existing as well as a small number of new API methods. Table 1 gives an overview of LSP support in different languages at the time of writing.

Central to our approach is our code execution strategy for dynamic analysis. For each feature, we detail both how information can be obtained statically and dynamically.

3.1 Obtaining Runtime Information

Code editor support for programming languages with static information about symbols, like Java, can be built by parsing and analyzing the source code. In dynamic languages with untyped symbols, however, there are only limited language-specific approaches to statically infer information about the code. These approaches are necessarily limited by certain dynamic language features (e.g. *eval* or results of foreign function calls). Furthermore, we are not aware of a performance-focused language implementation framework that supports static inference in a language-agnostic way for their implemented dynamic languages. The RPython framework does use type inference internally, but this cannot be applied to arbitrary dynamic languages developed with RPython.

Our basic approach to get runtime information at a specific position in the source code, is to execute that source code while observing the execution. We cannot rely on any language specifics to analyze execution or attach to it, and we do not want to burden the developer with having to do significantly more work that is not related to simply running the language.

Observing the execution is already possible in Truffle, because the framework provides a debugger. We can reuse the same approach to halt at an interesting location and obtain local symbols and their corresponding run-time values from the active frame. Details such as type, documentation, definition locations, and more can then be queried from Truffle objects.

However, executing arbitrary source code poses a number of challenges: we need to deal with side-effects, the possibility of a long running or non-terminating execution, or the fact that the source code of interested might not be reached by the execution flow.

Unreachable Source Sections There are likely source sections which are unreachable by the normal execution flow of the current source code file. Typically a program has a limited set of entry points, and only from those can other parts of the program be reached.

One entry point to reach a (hopefully) large number of source sections are unit tests. Furthermore, most unit tests have a short execution time and are supposed to be free of unwanted side-effects – or at least have a clean-up routine included. The systematic execution of unit tests is therefore a promising approach to get runtime information for code completion enhancement. This idea is inspired by *type harvesting* [4]. Programmers need to tell our language server how to execute the tests via a special comment in the first or last lines of a source code file, which encodes a path to a runnable script. Our approach heavily favors test-driven development, because it will produce better results with it. It also favors writing unit tests that are small, finish quickly, and cover only a small part of the code.

However, unit tests might not be available, in which case we fall back to executing the current source code that is being edited. One might question the scalability of this. We believe that users that write more complex programs also write unit test, because these are beneficial not only for our approach, but for development in general. However, as an alternative we could allow users to add comments to methods with example invocations similar to Babylonian programming [11] or Godoc examples[4].

Sandboxing the Execution Environment Users would not expect that the source code they are currently working on is executed when they trigger code completion, and they should not have to worry about side-effects in their code. Therefore, any execution needs to happen in a sandboxed environment.

Our language server, because it can observe all execution, is well positioned to provide such sandboxing. Furthermore, one of the goals of the Truffle framework is to be able to sandbox languages entirely, including all access to system resources. (RPython similarly can sandbox languages so that

[4]Changeset available at https://github.com/oracle/graal/pull/764

[1]https://github.com/graalvm/simplelanguage (accessed 2018-12-17)
[2]https://github.com/graalvm/graalpython (accessed 2018-12-17)
[3]https://github.com/oracle/fastr (accessed 2018-12-17)
[4]https://golang.org/pkg/testing/#hdr-Examples

Table 1. An overview over selected Truffle languages and the current implementation status of Graal-lsp features.

LSP feature	SimpleLanguage (SL)[1]	Python[2]	FastR[3]	GraalSqueak [9]
Symbols	✓	✓	✓	
Goto-definition (local variables)	✓	✓		
Goto-definition (callables)	★	★	★	?
References and Highlights	✓	✓		?
Syntax Errors	✓	✓		✓
Signature Help		★		
Hover Information	★	★	★	★
Completion (globals)	✓	✓	✓	✓
Completion (locals)	✓	✓	★	✓
Completion (properties)	★	★		✓

✓ Available statically
★ Available after collecting runtime information
? Unclear how to implement

all system calls are mediated through a trusted process.) We use the Truffle sandbox to disallow all native access, as well as all access to sockets or the file system and redirect such requests as well as possible. When writing to a file, for example, an in-memory file system is used as a layer over the real file system on demand. Reading from files then first checks the in-memory file before going to the actual file system.

Network I/O, signals, spawning new processes, or access to native libraries are not part of our approach and our server will just disallow such attempts. These limitations are difficult to lift in the general case, and we currently require well-written unit tests with mock objects to replace these operations to gather runtime information.

Unsaved Changes While editing, the source code is constantly out-of-sync with the actual file system. If we execute, we have to ensure that code imports do not hit the file system if our server has newer, but yet unsaved versions. The LSP specifies no request sent from server to client to trigger saving files.

As part of the sandboxing approach described above, we already provide an in-memory file system completely transparently to the languages and can serve the current state of unsaved files instead of the versions stored on the real file system.

Long Running or Non-terminating Executions The execution of arbitrary source code might take an unpredictable amount of time or might not finish at all. Therefore, our server immediately cancels execution when it has reached the desired position in the source code and we have collected the needed runtime information. Additionally, executions are canceled if they do not finish in a configurable amount of time. In case of a timeout, no runtime information can be provided to enhance our code completion.

Truffle's instrumentation framework allows our language server to observer coverage of statements. If the user has defined multiple unit test entry points, we plan to use coverage information to determine which tests to (re-)run in order to gather runtime information. This helps reduce the time required to produce results for the LSP client.

Caching Runtime Information Lag is in important factor in interactive code editors. Our approach for gathering information relies heavily on the runtime required for executing the tests. Since the server already initializes the languages and the Just-in-time (JIT) compiler optimizes unit test executions to some extent, the execution times get better over time, but in our examples, we frequently had to wait 1-2 seconds to run the relevant test and get a result. We have yet to gather experience with larger test suites with more than a few hundred lines of code.

When runtime information has been collected once e.g. for a function and the developer continues editing the function body, we can still use information about parameters gathered in previous runs. For this we keep a mapping from cached runtime information to source locations on the path to the execution where we gather the information. We only invalidate if any of those locations change.

3.2 Implementation of LSP Features

We will initially use SL as a running example to show how our approach was implemented for that language, and then extend it to other Truffle languages. SL is Truffle's exemplary language implementation, it has first class functions, floating-point numbers and strings as primitive types, and a single `object` type. Objects are implemented as simple key-value stores.

Like other Truffle tools, the LSP server is implemented as a `TruffleInstrument`. Figure 1 gives an overview of how it interacts with the Truffle framework. Instruments can ask for notifications in response to various events such as the creation of AST entry points (*call targets*) or the pending execution of an AST `Node` via one of its `execute*` methods. An instrument can also trigger parsing or execution via a `TruffleLanguage` instance, change the execution flow, and interact with any `TruffleObject` that flows through the AST. Finally, an instrument can influence the environment from which languages get access to OS-level facilities including files and thus inject a custom `FileSystem` to serve requests for file access from a sandboxed implementation rather than the real system.

Specific to our instrument is that it launches an HTTP server on creation. This `LSPServer` handles the client's LSP requests. The handlers for the various requests are where the features we describe in this section are implemented. Whenever a file is touched, the `didOpen` and `didChange` callback uses the languages to `parse` the source. This immediately provides syntax errors and records any created AST nodes. These recorded nodes are then used in subsequent requests to map from editing locations to the AST, by filtering on the results of `getSourceSection`.

Navigational Features

Querying Symbols The *documentSymbol* and *symbol* requests of the LSP ask for known names and their locations in the source code. For functions, this already works for all file-based Truffle languages. After parsing, our instrument was notified of any *root* node (e.g. of a function) and can use its `getSourceSection` method. The language developer has to implement this method, but this is already required for other Truffle tools and thus did not need additional support.

We have extended the list of tags that Truffle uses to identify the purpose of AST nodes with a `DeclarationTag`. For SL, we needed to extend the `hasTag` method in those nodes that introduce variables to respond true for that tag to have them show up in the list of symbols. For additional information, we also made use of the `getNodeObject` API provided by Truffle nodes. The convention we chose is that, if the language developer returns an object from this method for a node tagged as a declaration, that object can respond to the `kind` message with more detailed information about the declared variable. This allows languages to communicate statically known types if possible. For SL, we here return the primitive type or `object` if the RHS of the assignment is a constant.

Go to Definition When the user selects a function name and asks for the *definition* location, we attempt to provide a static location or a dynamic one. In the static case, we only need to search the AST for a node representing the function call. These are already tagged in SL with the `CallTag` to support the Chrome debugger. From that node we match the name of the function call to function definitions that have been parsed to show a list of candidates. Similarly, given our newly introduced `DeclarationTag`, we can also query the definition locations of variables with a certain name.

If we are able to gather runtime information, we can provide more accurate information. To gather runtime data, we send another `parse` request to the language and execute the resulting call target. This parse request either contains a unit test or some other entry point declared by the user, or just the current source file as described in subsection 3.1. Without runtime data, we fall back to calling `findTopScopes` and looking for the name there.

We use the `TruffleLanguage\#findSourceLocation` API, which language developers can implement to provide the definition location of objects where it is convenient to provide. For most languages including SL, function objects are supported by this API and we can thus unambiguously determine the definition of a function call even when there are multiple functions with the same name.

Finding References For the *documentHighlight* and *references* LSP messages, we re-use the `CallTag` and `DeclarationTag` annotations to find nodes with the names we are interested in. Furthermore, we have added two more tags, `ReadVariableTag` and `WriteVariableTag`, which we have added to nodes representing local variable accesses in SL. Together, these allow us to return references for variables and functions.

To take scopes into account, we make use of another Truffle API that is already implemented in SL, `findLocalScopes`, which enumerates scopes from the inside out. If, for example, the client asks to highlight all references to a variable in a closure, the local scope API will enumerate surrounding scopes up to the top scope. Our server can then search for the same names in those scopes and find the correspondingly tagged nodes. This API can be used statically or with a run-time `Frame` object, and returns lexical scope or dynamic scope nesting, respectively.

Informational Features

Detecting Syntax Errors Language runtime parsers are not usually built to support incremental parsing or detection of more than one syntax error. SL, like other Truffle languages, fails parsing at the first syntax error, so that only one error per file can be sent to the client. By convention, a language syntax error is supposed to implement the `TruffleException` interface (not shown in the diagram), which provides a `getSourceLocation` method. Figure 2 displays how such a diagnostics notification is visualized in SL.

Documentation and Signature Help The *signatureHelp* request is auto-activated at client-side, like code completion, if the server includes trigger characters for signature help in the *initialize* response. A *signatureHelp* response consists of a list of signatures, because there might be different variants, if the language implementation supports function overloading.

If we can gather runtime information to access the object representing the function on the Truffle level, we have added

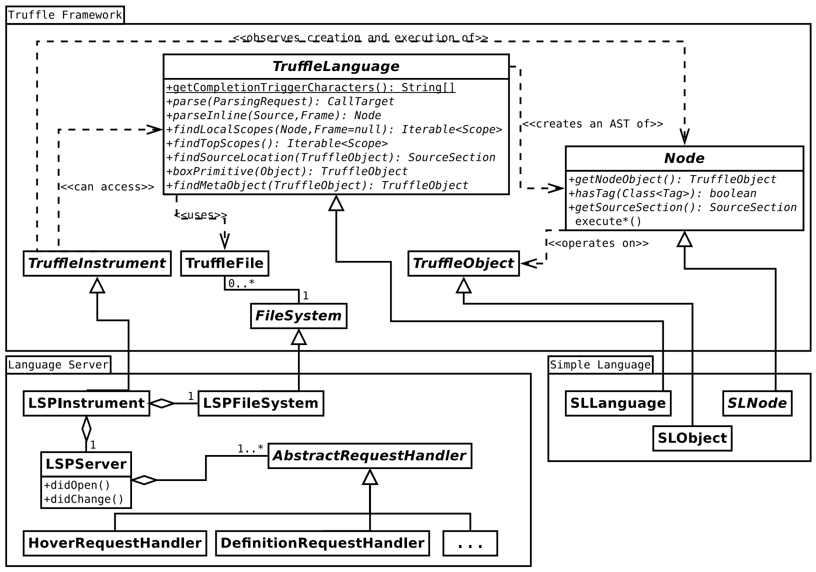

Truffle Framework
<<observes creation and execution of>>
TruffleLanguage
+getCompletionTriggerCharacters(): String[]
+parse(ParsingRequest): CallTarget
+parseInline(Source,Frame): Node
+findLocalScopes(Node,Frame=null): Iterable<Scope>
+findTopScopes(): Iterable<Scope>
+findSourceLocation(TruffleObject): SourceSection
+boxPrimitive(Object): TruffleObject
+findMetaObject(TruffleObject): TruffleObject
<<creates an AST of>>
Node
+getNodeObject(): TruffleObject
+hasTag(Class<Tag>): boolean
+getSourceSection(): SourceSection
execute*()
<<can access>>
<<uses>>
<<operates on>>
TruffleInstrument
TruffleFile
TruffleObject
0..*
1
FileSystem
Language Server
LSPInstrument
1
LSPFileSystem
1
LSPServer
+didOpen()
+didChange()
1..*
AbstractRequestHandler
HoverRequestHandler
DefinitionRequestHandler
...
Simple Language
SLLanguage
SLNode
SLObject

Figure 1. Simplified class diagram of LSP components, Truffle, and SL.

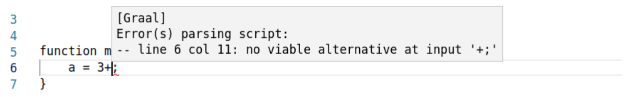

```
3                [Graal]
4                Error(s) parsing script:
5   function m   -- line 6 col 11: no viable alternative at input '+;'
6       a = 3+;
7   }
```

Figure 2. Diagnostics used to display a syntax error in SL.

two API messages that the language developer needs to implement: `GetDocumentation` and `GetSignature`. These should return a documentation string or all parameter names and their documentation, respectively. The formatting of these is language-specific and left to the language developer.

If we either cannot gather runtime information or the additional API is not implemented by the language, we fall back to trying to find the definition of the function (as described above) and show the surrounding source code as a hint to the developer.

SL does not keep track of documentation strings in its parser, and neither does the parser keep parameter names separate from local variables. Implementing these two messages will require more work in the parser of SL.

Hover Information We can support the *hover* request to show information for variables and callables with type information, provided there is runtime information available. If we have found an AST node representing a variable or callable at the requested position, we use `parseInline` and evaluate the resulting node. With the result, we build the hover response. Most Truffle languages including SL already implement the `findMetaObject` API. We use it on the result to show the type here in addition to the concrete value.

Completion

We distinguish two kinds of code completions which need different levels of support by the language: completion of globals or locals and completion of properties.

Globals and Locals We use the code snippet in Listing 1 to illustrate how to support code completion of globals and locals. With the caret on line 3, a *completion* request is sent to the server for line 3 at character offset 4. The server searches parsed nodes for the one with the nearest `SourceSection` and uses it to call `findTopScopes` and `findLocalScopes`. We would expect a completion result to include items for all SL globals (e.g. built-in functions) and the locals `param1`, `param2`, and `x`. The variable `y` should not be included, because no value is assigned to it until line 4.

Statically, we get the names and values of SL built-ins in the top scopes. The values are `SLObjects`, so we can use the normal `TruffleObject` messages to query if they are *callable*, *instantiable*, and so on and return appropriate flags to the client. In languages that respond to documentation and signature messages for their objects, we show that here as well, but there is no such support for SL. For `findLocalScopes`, SL statically returns the names of *all* variables defined.

In this case, we can just execute the entire file to gather run-time data. As soon as the execution reaches the node we are interested in, we stop and use the current live `Frame` in the `findLocalScopes` method. Now we get the local scope including live locals, we can show the types for `x` and the parameters and we reject `y`, because it is `null` at this point.

Property Completion Listing 2 shows a code snippet where we define a `Natural` object with with a variable `n` and a method `next`. In the second-to-last line, we assume that the dot was the last character inserted and thus a code completion request was triggered. A completion result should

Listing 1. Completions in SL code.

```
function foo(param1, param2) {
    x = 42;
    _
    y = param1 + param2;
}
function main() { foo("1", "2"); }
```

Listing 2. SL code snippet stopped at property completion.

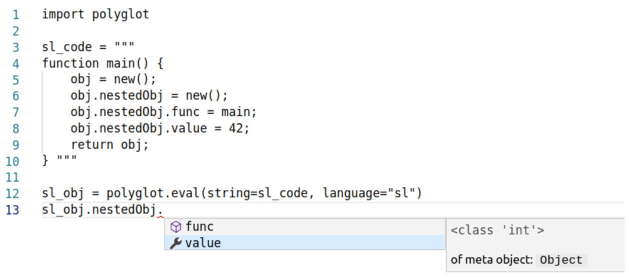
```
function next(self) {
    r = self.n;
    self.n = self.n + 1;
    return r;
}
function Natural() {
    newobj = new();
    newobj.n = 2;
    newobj.next = next;
    return newobj;
}
function main() {
    natural = Natural();
    natural.
}
```

include `n` and `next` as properties of the object referenced by the symbol `natural`.

To support this kind of code completion, we need runtime information. We also have to detect that an object property completion is needed instead of a completion of globals and locals. Second, the source code will typically contain a syntax error, when this kind of completion request is sent. Third, we have to identify the AST element for which the completion of properties is desired. Fourth, we have to obtain the actual object associated with that AST-element, which might be available only at run-time. Finally, we need to get the object's properties to create the completion items.

As prerequisite for completion of properties, we need to query the language implementation for its *completion trigger characters*. We have added the method `getCompletionTriggerCharacters` and language developers need to return a list of strings which should trigger completions. For SL, this is only the dot.

When a completion request arrives, the location is preceded by a completion character. This often means that the source is not parseable. Our current approach is to track the most recent modifications the client sent us since the last parseable state and use this state to execute the code. In many cases, this will simply be the state before the completion trigger character was inserted. In our example, just removing the last dot makes the code parseable again and we end up with the correct object to provide completion for. However, in some cases multiple edits have taken place in different parts of the source, and no parseable state can be reached this way that also leads to the desired code location. (We regard this problem as orthogonal to our approach, but it needs further investigation.)



```
import polyglot

sl_code = """
function main() {
    obj = new();
    obj.nestedObj = new();
    obj.nestedObj.func = main;
    obj.nestedObj.value = 42;
    return obj;
} """

sl_obj = polyglot.eval(string=sl_code, language="sl")
sl_obj.nestedObj.
```

```
<class 'int'>
of meta object: Object
```

Figure 3. Completion of properties for a foreign SL object in Python source code.

We search for a matching AST node in front of the (removed) completion trigger character position by checking the source sections. We are only interested in the last node which represents an expression, because we want to obtain its result. Truffle's node tagging allows use to query for the `ExpressionTag` class to obtain only the expression `natural` in our example.

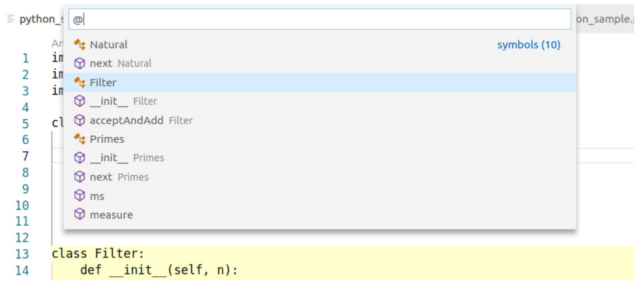
For literals, the AST nodes can return the literal object directly for static completion via the `getNodeObject` method, where they should return a Truffle object with a member named `literal`. In our example, `natural` is not a literal, so we evaluate the expression node using `parseInline` in the current frame. If no run-time data is available, we attempt to evaluate in the global scope using `parse`. This way, at least completions on scope-independent objects can be satisfied.

Finally, if the returned object is not a `TruffleObject` but a primitive type, we need to box it to read the members the language assigns to these types, if any. We defined the API method `boxPrimitive` and implemented it in `SLLanguage`. Now we get a `SLObject` representing the primitive value and can query it for members.

Polyglot Code Editing Using SL code embedded in Python, we want to highlight how Truffle's support for polyglot programming works naturally with the features described above. Figure 3 shows Python code evaluating a SL code snippet which creates and returns a SL object. In the last line, we get completions of properties for the foreign SL object. This demonstrates how the language-agnostic approach of our server does not depend on the language an object originates from. After implementing our approach for SL, this example worked immediately after *only* adding the completion trigger character API to Python.

3.3 Application to Other Languages

In this section, we show by means of the Truffle language implementations Python, FastR, and GraalSqueak that basic support of some LSP features can be achieved with minimal implementation effort in real languages. Furthermore, we explain which LSP features cannot be easily supported in some language implementations and other limitations of our approach.



```
class Filter:
    def __init__(self, n):
```

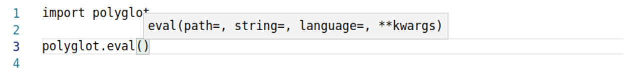
Figure 4. Symbol overview over the current Python document. The selected entry's source section is highlighted in the editor.

Evaluated Truffle Languages The Python implementation for GraalVM was modified to implement all API calls we proposed. For FastR, the implementation of the R[5] language, we have only implemented support for completion trigger characters, so the implementation is nearly unchanged. This demonstrates that we can provide some support even with the least amount of language developer effort. GraalSqueak [9], a Squeak/Smalltalk [6] implementation, is image-based rather than file-based and executes bytecodes rather than a Smalltalk AST. Therefore, it is difficult to map concepts of GraalSqueak to LSP. We have chosen it to show that at least some features can be supported with reasonable effort. We treat every opened `st` file as an isolated Squeak/Smalltalk scope.

Navigational Features Both Python and FastR automatically support querying global symbols and navigating to them. For Smalltalk, this feature is not available, because the global symbols are not defined in any source file, but are part of the image. Thus, while we can list global symbols, we cannot offer a source location for them, which is required for the highlighting that LSP clients want to support, as seen in Figure 4.

To jump to definitions, we implemented the new `DeclarationTag` in Python to support jumping to the definition of local variables. For FastR, we did not change anything, but since the language supports returning the source location for functions, jumping to the definition of functions when runtime information is available worked immediately. In GraalSqueak, the `DeclarationTag` could enable jumping to the definition of local variables, but it is unclear how to implement jumping to function definitions at all. Furthermore, GraalSqueak works on bytecode, yet in Smalltalk source code, variables are declared explicitly at the beginning of blocks or methods. These declarations, however, are not available in the bytecode, where variables are only referenced by index. Thus, the Smalltalk implementation would have to do significant work to map back from AST nodes representing bytecode locations to source locations.

[5]https://www.r-project.org (accessed 2018-12-17)

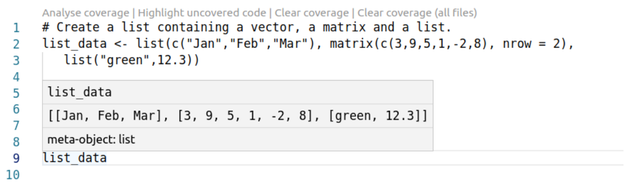


```
import polyglot

polyglot.eval()
```

```
eval(path=, string=, language=, **kwargs)
```

Figure 5. Signature help for Python.


Analyse coverage | Highlight uncovered code | Clear coverage | Clear coverage (all files)

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),
   list("green",12.3))

list_data
```

```
list_data
[[Jan, Feb, Mar], [3, 9, 5, 1, -2, 8], [green, 12.3]]
meta-object: list
```

Figure 6. Displaying information on mouse hover for FastR source code using cached runtime information.

To fully support references, we had to add the `Write/ReadVariableTag` in appropriate places to Python. While FastR uses the `CallTag`, and thus we can provide references and highlights for functions, we cannot do the same for local variables. Additionally, scoping information in R relies on the dynamic execution extent, and thus we cannot properly support these features on FastR without both the tags and runtime information.

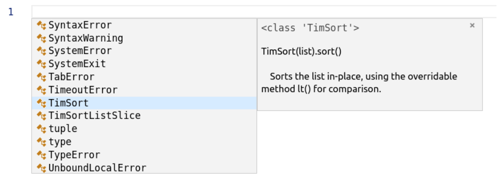
Informational Features All languages except FastR fail parsing at the first syntax error, so that only one error per file can be sent to the client. FastR does not throw a `TruffleException` in case of a syntax error, so that GRAALLSP cannot report any syntax errors for FastR. However, this is arguably a bug in the FastR implementation, as it also affects other tools such as the debugger.

Signature help and documentation requests are only fully implemented in Python. These use our newly introduced `GetSignature` and `GetDocumentation` messages. In Python, these were easy to implement since functions already store and expose their documentation to user code (Figure 5). For the implementation of these messages we thus just had to map them to the appropriate Python code. For Smalltalk classes and functions, we implemented the `GetDocumentation` message to show the last modified date, category, and author; these are stored as meta-data on these Smalltalk objects anyway.

Finally, hover information is available to the same extent as for SL in all three other languages, since they each support dynamic evaluation of source code in a given runtime frame. An example for FastR is shown in Figure 6.

Completion Figure 7 shows a code completion for globals and locals in Python. Completion items have different icons for classes, methods, and variables in the list of suggestions. For the selected item of a built-in class, a string representation of the corresponding meta-object is displayed, together with a documentation, if available. The same support is available for FastR and GraalSqueak (Figure 8).

The code completion of properties is supported to different extents in these languages. Runtime information are



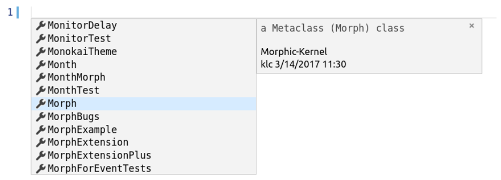
```
SyntaxError
SyntaxWarning
SystemError
SystemExit
TabError
TimeoutError
TimSort
TimSortListSlice
tuple
type
TypeError
UnboundLocalError
```

```
<class 'TimSort'>

TimSort(list).sort()

   Sorts the list in-place, using the overridable
method lt() for comparison.
```

Figure 7. Completion of globals with documentation in Python.



```
MonitorDelay
MonitorTest
MonokaiTheme
Month
MonthMorph
MonthTest
Morph
MorphBugs
MorphExample
MorphExtension
MorphExtensionPlus
MorphForEventTests
```

```
a Metaclass (Morph) class

Morphic-Kernel
klc 3/14/2017 11:30
```

Figure 8. Code completion of globals in GraalSqueak.

always required due to the dynamic nature of the languages. For FastR and GraalSqueak, we were not able to provide a completion of properties yet, although the Truffle API allows it. These languages are missing an expression annotation on their AST nodes. Implementing it will not only help the LSP support, but also add support for column breakpoints in the Chrome debugger, for example.

In Python, triggering a completion of properties for the code snippet `anObject.nestedObject`, by inserting a dot at the end, will yield completion items for all properties of the object referenced by `anObject.nestedObject`, because the nearest node tagged as expression is the one representing the whole source section. In SL, however, inserting a dot before the semicolon in the expression `anObject.nestedObject;` to trigger code completion, will yield no results, because the nearest node tagged as expression represents just the `nestedObject` string, and not `anObject.nestedObject`. This highlights that the developer has to take a bit of care designing their AST for optimal developer experience.

4 Related Work

Our approach focuses on enabling performance-focused language implementation frameworks to provide their implemented dynamic languages with IDE-like tooling support with minimal effort. Another major part of our approach is based on collecting and using runtime information to enhance certain LSP features.

IDE-frameworks such as Eclipse[6], Visual Studio[7], or IntelliJ[8] provide extension points to develop IDE-specific plugins for new languages or can be used as language clients if they implement the LSP. But as they do not support language developers in implementing programming languages in the first place and writing IDE-specific plugins or a language server still requires a lot of work, we do not further discuss them here. Furthermore, there are lots of language- and editor-specific tooling solutions for dynamic languages, which are also not our focus.

Language Workbenches such as like Spoofax [7], Xtext [3], or MPS [20] focus foremost on the efficient definition, reuse, and composition of languages and their IDEs [2]. In contrast to the work presented here, performance and generality of the resulting languages is a secondary concern.

Visser et al. described an approach using the Spoofax language workbench [7] and the DynSem language [18] to systematically generate Truffle languages, i.e. AST interpreters, from DynSem specifications [19]. However, Spoofax provides plugins for the Eclipse IDE only. It would be interesting to see if LSP support can easily be enabled for Truffle languages generated in this way.

Two approaches that focus on reusing existing tools of a live programming environment and the overall programming experience, are the Smalltalk-based IDEs Helvetia [12] and Squimera [10]. Helvetia enables embedding DSLs into Smalltalk, by adding hooks into the Smalltalk compiler chain. This way, the DSLs can be translated on demand into Smalltalk code, so that existing tools of the environment still work, without the need to generate and start a new IDE. Squimera is based on a multi-language runtime and allows reuse of the Smalltalk tools across all supported languages. In contrast to this work, these approaches focus on Squeak/Smalltalk as the DE.

Part of our approach focuses on collecting and utilizing runtime information to answer LSP requests. There are a number of prior approaches towards this idea.

The Hermion IDE [14] collects run-time information to enhance program comprehension by improving static source code navigation and browsing in Squeak/Smalltalk. Hermion builds upon partial behavioral reflection [15] to restrict the collection of runtime information to certain source code parts and detail levels. However, users need to manually specify which classes shall be instrumented to keep the cost of instrumentation and reflective operations at an acceptable level. In Graal-lsp, the set of instrumented AST nodes is implicitly defined by the LSP workspace, i.e. all parsed source code files. As Hermion continuously observes the instrumented classes, it can collect different types for a variable, which the current Graal-lsp implementation does not do.

Holmes and Notkin [5] described an approach to improve the static find-references feature of the Eclipse IDE using collected runtime information. Only executed methods are included in the search results, so that users benefit from a limited result set related to their current application. To collect the required runtime information, they instrument the application's tests with AspectJ [8]. This is comparable to the case when run-time data is available in our approach.

[6]https://www.eclipse.org/ (accessed 2018-12-17)
[7]https://visualstudio.microsoft.com (accessed 2018-12-17)
[8]https://www.jetbrains.com/idea (accessed 2018-12-17)

Type harvesting is another approach to collect type information for dynamic programming languages [4]. The solution is based on observing the types of variables during test execution. A similar feature exists in PyCharm[9] which can collect runtime information during a debug session. Graal-lsp has the advantage of running with the JIT even when instrumented, yielding better performance than these approaches.

5 Conclusion and Future Work

We described the problem of language implementation frameworks lacking tooling support for code editing, especially for implementations of dynamic programming languages. Language developers that use such frameworks to create and maintain programming languages with reasonable effort and good performance currently get at most some run-time tooling. We propose an approach to have IDE-like code assistance with minimal work required from the language developers. Our approach is based on a language-agnostic LSP server and the collection and utilization of runtime information. As proof of concept, we have implemented Graal-lsp, a language server for the GraalVM.

There are three limitations of our approach to consider especially: First, since most runtime parsers stop on the first syntax error, we cannot report more than one syntax error per file to the client. Second, all LSP features rely on the availability parseable source. Third, when our sandboxing prevents some kind of access or triggers a timeout, some source sections of interest may not be covered and thus no or only incomplete information is available.

Future work is two-fold: the main work will be to address the above limitations and find ways around them for realistic scenarios. This includes measuring how much time we can afford to spent executing code before the user gives up waiting for a result. The other is to cover more LSP features in our approach. Part of this means to investigate further how languages such as Squeak/Smalltalk, which use bytecodes as part of the execution model and do not map to files, can be made to work with LSP.

Despite these limitations, we already feel that our approach leads to a quickly usable code editor for the language user. Since we use the instrumentation API of the Truffle framework to parse and execute source code and answer LSP requests, no separate parser or inference engine is required. Sandboxing avoids unwanted side-effects when executing arbitrary source code. As by-product of using the Truffle framework, Graal-lsp supports polyglot editing assistance. With only few API extensions for Truffle languages to implement we demonstrated that a basic set of important LSP features can be supported with minimal implementation effort and we are confident that a similar approach can work with reasonable effort in other, similar frameworks such as RPython.

[9]https://www.jetbrains.com/pycharm/ (accessed 2018-12-17)

Acknowledgments

We gratefully acknowledge the financial support of Oracle Labs[10], HPI's Research School[11], and the Hasso Plattner Design Thinking Research Program[12].

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. 2007. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS)*. ACM, 53–64. https://doi.org/10.1145/1297081.1297091

[2] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[3] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) 2010*. ACM, 307–309. https://doi.org/10.1145/1869542.1869625

[4] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. 2011. Type Harvesting: a Practical Approach to Obtaining Typing Information in Dynamic Programming Languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1282–1289. https://doi.org/10.1145/1982185.1982464

[5] Reid Holmes and David Notkin. 2010. Enhancing Static Source Code Search with Dynamic Data. In *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*. ACM, 13–16. https://doi.org/10.1145/1809175.1809179

[6] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *SIGPLAN Notices*, Vol. 32. ACM, 318–326. https://doi.org/10.1145/263700.263754

[7] Lennart CL Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA) 2010*, Vol. 45. ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*. Springer, 327–354. https://doi.org/10.1007/3-540-45337-7_18

[9] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ACM, 30–35. https://doi.org/10.1145/3242947.3242948

[10] Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. 2018. Live Multi-language Development and Runtime Environments. *The Art, Science, and Engineering of Programming* 2, 3 (mar 2018). https://doi.org/10.22152/programming-journal.org/2018/2/8

[10]https://labs.oracle.com/
[11]https://hpi.de/en/research/research-school.html
[12]https://hpi.de/en/dtrp/

[11] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *CoRR* abs/1902.00549 (2019). arXiv:1902.00549 http://arxiv.org/abs/1902.00549

[12] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. 2010. Embedding Languages Without Breaking Tools. In *European Conference on Object-Oriented Programming*. Springer, 380–404. https://doi.org/10.1007/978-3-642-14107-2_19

[13] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, 944–953. https://doi.org/10.1145/1176617.1176753

[14] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. 2008. Exploiting Runtime Information in the IDE. In *Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008. ICPC 2008*. IEEE, 63–72. https://doi.org/10.1109/ICPC.2008.32

[15] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. 2003. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. *ACM SIGPLAN Notices* 38, 11 (2003), 27–46. https://doi.org/10.1145/949305.949309

[16] Michael L Van De Vanter. 2015. Building Debuggers and other Tools: We Can Have it All. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ACM, 2. https://doi.org/10.1145/2843915.2843917

[17] Guido van Rossum. 1995. *Python Tutorial*. techreport CS-R9526. Centrum voor Wiskunde en Informatica (CWI).

[18] Vlad Vergu, Pierre Neron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications (RTA) 2015 (Leibniz International Proceedings in Informatics (LIPIcs))*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 365–378. https://doi.org/10.4230/LIPIcs.RTA.2015.365

[19] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 95–111. https://doi.org/10.1145/2661136.2661149

[20] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012*. IEEE, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[21] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 13–14. https://doi.org/10.1145/2384716.2384723