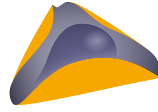


On Demand Runtime Information

A language- and IDE-agnostic approach to provide runtime information

Master Thesis

Rathesan Iyadurai
from
Ersigen BE, Switzerland

University of Bern

January 2020

Prof. Dr. Oscar Nierstrasz
Dr. Haidar Osman
Dr. Boris Spasojević

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Understanding programs written in dynamically-typed languages can be difficult because of the lack of static type information. When reasoning about a function written in JavaScript for instance, developers often lack the information about parameter types and values, which is essential for understanding the implementation. Gathering runtime information and presenting it on-demand would assist developers in their program comprehension tasks.

In this thesis we present AUDREY, a system that gathers runtime information for multiple languages and exposes the information to many development environments. AUDREY aims to be as language- and IDE-agnostic as the underlying infrastructure allows. We discuss the challenges of implementing AUDREY with state-of-the-art technologies for future work.

Contents

1 Introduction **3**

2 Related Work **6**

3 Technical Background **8**
- 3.1 Truffle . . . 8
- 3.2 Language Server Protocol . . . 9

4 Audrey in a Nutshell **12**
- 4.1 Design Rationale . . . 13

5 AUDREY Implementation **14**
- 5.1 AUDREY Sampler . . . 14
- 5.2 AUDREY Server . . . 18

6 Discussion **20**
- 6.1 Finding an IR for language-agnostic instrumentation . . . 20
- 6.2 The missing tooling infrastructure . . . 21
- 6.3 Instrumentation overhead . . . 22
- 6.4 Information leakage . . . 24

7 Future Work **26**

8 Conclusions **28**

A Installing and Using `Audrey` **30**
- A.1 Installation . . . 30
  - A.1.1 Prerequisites . . . 30
  - A.1.2 Installing `Audrey` . . . 31
- A.2 Usage . . . 31
  - A.2.1 Gathering runtime information with `Audrey` . . . 31
  - A.2.2 Displaying runtime information in a development environment . . . 32

1
Introduction

Program comprehension and software evolution in dynamically-typed languages such as Ruby or JavaScript can be a mentally straining endeavor [2, 11]. For instance, to comprehend which types a Ruby method takes as parameters, developers are tasked with tedious actions that do not always yield precise answers. They could attempt to infer the types from looking at the parameter names. They could scan the method body to identify messages being sent to the parameters in order to make a more precise inference. They could try to look at the call sites in order to find some example arguments being passed in, potentially resulting in deeper searches or non-representative examples in test code. They could inspect the state of a particular execution using a debugger. If they are lucky, their team maintains an up-to-date per-method documentation with type annotations and examples in a standardized format, such as YARD.[1] Mitigating these issues in dynamically-typed languages has been heavily researched in the past couple of decades, *e.g.,* by relying on parameter naming conventions [23] or employing various type inference strategies [1, 3, 4, 12–14, 16–18, 22].

One reason why program comprehension and software evolution tasks are so tedious is the lack of static analysis tools such as those available for statically typed languages such as Java or C#. Even in medium sized code bases, a task as simple as renaming a Ruby method and all its senders is daunting, oftentimes requiring developers to perform manual search and replace actions. The dynamic nature of languages like Ruby and JavaScript complicates the implementation of tools that could automate such tasks [8].

We believe that the key to understanding a program written in these languages is the runtime. For instance, thanks to the dynamic nature of Ruby and JavaScript, most language constructs can be queried at run time. Hence, the truth about the shapes and values of objects, *e.g.,* real examples of what is being passed in as function parameters in JavaScript, can only be found in a running program.

In this thesis, we describe a strategy to gather just enough information from a running program, so that development environments can be enhanced in a way that makes the aforementioned program comprehension tasks less tedious, *e.g.,* by providing type information and a textual representation of example arguments in the IDE.

[1] `https://yardoc.org`

Consider the JavaScript function in Listing 1.1 that is part of a raytracer[2] project.

Listing 1.1: A JavaScript `blend` function

```
function blend(c1, c2, w) {
  var result = new Color(0,0,0);

  result = add(
    multiplyScalar(c1, 1 - w),
    multiplyScalar(c2, w)
  );

  return result;
}
```

It is difficult to reason about the behavior of the function `blend` given the lack of information on the parameters and return values. An attempt to acquire more information by studying invocations of `blend` or usages of the parameters in the function body, *e.g.,* as parameters of `multiplyScalar`, requires additional actions by the developer and is not guaranteed to yield precise answers.

While it is difficult to reason about the shapes and values of the parameters of `blend` from a static perspective, the running raytracer at one point had concrete examples. Our strategy allows developers to access the runtime information on-demand, *e.g.,* by being presented with textual representations and concrete type information of example arguments as seen in Figure 1.1 when hovering over `blend`.

Figure 1.1: Example runtime information when hovering over `blend` in Listing 1.1

In this particular example, the runtime information served immediately reveals that the function `blend` is operating on some sort of color objects with `red`, `green` and `blue` properties.

As many languages and development environments suffer from the afore-mentioned issues, our strategy strives to be as language- and IDE-agnostic as the underlying infrastructure allows it. We have set ourselves three main requirements, which can be summarized as following:

[2] `https://chromium.googlesource.com/v8/v8/+/4.3.18/benchmarks/raytrace.js`

- The strategy is language agnostic, i.e., it can be reused for many languages with little to no engineering effort.
- The strategy is development environment agnostic, i.e. it can be reused for many editors and IDEs with little to no engineering effort.
- The runtime information can be gathered with minimal instrumentation overhead, so that it is practical to do so in a production environment.

We deliberately formulate these requirements in an idealistic manner, as our intention is to explore the challenges and limitations of state of the art technologies when it comes to fulfilling these requirements. More concretely, these requirements pose the following research questions that we explore and discuss in this thesis:

- What is a practical underlying run-time representation of a running program that allows us to gather the information for as many languages as possible?
- How can we keep the instrumentation overhead to a minimum while gathering the information?
- How do we prevent gathering sensitive information?
- How do we expose the information to various development environments without writing multiple IDE-specific plugins?
- How do we retrieve the relevant runtime information for a given source location in the development environment?
- How do we keep the stored information up to date when the source code is modified?

As a proof-of-concept for our strategy, we present AUDREY (titivation of the abbreviation ODRI for **O**n **D**emand **R**untime **I**nformation), an implementation that leverages the built-in instrumentation capabilities of Truffle, a framework for building languages in the form of AST interpreters in Java. AUDREY collects concrete type information and textual representations of run-time values for any language implemented using the Truffle framework (Truffle languages) without having to implement any language-specific behavior. To avoid tailoring to a specific IDE's infrastructure, AUDREY serves the gathered information over the language server protocol (LSP).[3] LSP is a protocol that standardizes communication between a development environment (*e.g.,* a simple source code editor) and a server that implements language-semantics-aware features such as autocompletion.

As these are fairly recent technological developments, we provide more background on Truffle and the LSP in chapter 3. In chapter 4, we present the architecture of AUDREY and the reasoning behind our design decisions. In chapter 5, we describe in more detail how AUDREY is implemented using Truffle and the LSP to remain language- and IDE-agnostic. In chapter 6, we discuss the major challenges we encountered while attempting to fulfill the aforementioned idealistic requirements, how well AUDREY currently addresses the challenges and what technical limitations currently prevent us from completely fulfilling the requirements. We propose some immediate future work in chapter 7 and conclude this thesis in chapter 8.

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

2
Related Work

Incorporating dynamic analysis to enhance static analysis based developer tooling is not a novel idea and there has been a substantial amount of research conducted in this area. To the best of our knowledge, AUDREY is the first system that focuses on providing runtime information language- and IDE-agnostically.

Röthlisberger *et al.* enhanced the browsing and navigation capabilities of the Squeak Smalltalk IDE by incorporating runtime type information [20]. They use a Smalltalk framework called Geppetto [7, 19] to hook into certain events of Squeak's bytecode interpreter, *e.g.,* message sends, in order to collect runtime information. The collected information is then used to improve program comprehension in the IDE by providing type information or improving the precision of navigational mechanisms.

A large amount of inspiration for AUDREY comes from the research conducted in dynamic type inference. Milojković *et al.* exploit the type information stored in a JIT compiler's inline caches to improve the precision of type inference [15]. This approach is particularly interesting to us, as we could also profit from GraalVM caches and reduce the instrumentation overhead when it comes to gathering type information only.

Wilkinson's Live Typing automatically annotates Smalltalk variables based on what objects were assigned to them [25]. Live Typing focuses on providing fast feedback during the development phase, *i.e.,* the variables are re-annotated quickly as the IDE and the running program use the same underlying OpenSmalltalk-VM.[1] This is achieved by modifying the bytecode instructions that store objects to additionally keep the types for later reference by the IDE.

We strive to provide the same tooling benefits as the approaches above, but have set ourselves three key requirements that separate AUDREY from existing approaches:

[1] `https://github.com/OpenSmalltalk/opensmalltalk-vm`

- We want to gather runtime information for all languages. Ideally, we have one tool that can be used to instrument all languages with little to no adjustments.
- We want to support all development environments, be it as simple as a text editor or a fully fledged IDE. Everyone should be able to integrate the gathered runtime information.
- The runtime information can be gathered with minimal instrumentation overhead, so that it is practical to do so in a production environment. This is an important requirement, as this makes gathering realistic data more practical.

These three requirements are deliberately idealistic. Our intention is to explore the challenges and limitations of state of the art technologies when it comes to fulfilling these requirements.

3 Technical Background

In this chapter we provide a brief overview of the technologies at play within AUDREY. In section 3.1, we describe the Truffle language implementation and instrumentation framework. In section 3.2, we illustrate the language server protocol.

3.1 Truffle

To gather useful information for multiple languages with one approach, we require these languages to share a common underlying representation that can be instrumented. Say we want to gather run-time examples with type information for method arguments in Ruby and function arguments in JavaScript with the same approach. Although both of these languages implement callable constructs with similar semantics, we lack a common instrumentable intermediate representation to avoid repeating the implementation effort for both languages considering their default implementations. This is where the Truffle language implementation and instrumentation frameworks come into play.

The Truffle framework allows languages to be expressed as self-optimizing AST interpreters written in Java (the host language), thus sharing a common underlying representation: the Truffle AST. Languages implemented using Truffle (Truffle languages) all target GraalVM, a variant of the Java Virtual Machine (JVM) that enables just-in-time (JIT) compilation of Truffle ASTs resulting in high performant languages [26]. Consequently, Truffle languages have one layer describing the semantics of the language (the AST interpreter) while the underlying VM can derive optimized versions.

At the time of writing, there are various Truffle implementations of popular languages that aim to be a drop-in replacement of their reference implementations, such as FastR[1], TruffleRuby[2], GraalJS[3] or GraalPython.[4] This makes the Truffle AST a particularly promising IR to build tools upon [24].

Another notable feature is Truffle's cross-language interoperability capabilities. Truffle comes with a language-agnostic mechanism for efficiently calling foreign-language-specific operations at run time [9].

[1]https://github.com/graalvm/fastr
[2]https://github.com/graalvm/truffleruby
[3]https://github.com/graalvm/graaljs/
[4]https://github.com/graalvm/graalpython

This mechanism is also leveraged by tool developers to delegate language specific tasks to the respective language implementation, *e.g.,* retrieving string representations or looking up type information, thus supporting substantially more language-agnostic tooling.

Most importantly for our purposes, Truffle comes with an instrumentation API that enables instrumenting various events in the Truffle AST by rewriting the AST at run time [6]. For instance, the API enables insertion of custom nodes after an AST node that will be notified when the node has finished executing. The API relies on Truffle languages tagging interesting nodes, *e.g.,* return statements, to profit from the instrumentation capabilities. The details of this tagging mechanism are outside the scope of this work. For our purposes, it is sufficient to understand that Truffle languages provide these tags and that the instrumentation framework allows us to inject custom behavior when certain tags are encountered.

In summary, Truffle enables tool building for multiple languages by injecting custom nodes into the Truffle AST. These custom nodes will be notified about various events at the inserted location. The mechanism is kept language-agnostic by relying on language implementers to tag nodes of interest.

3.2 Language Server Protocol

Traditional development environments provide the language specific tooling themselves or come with third-party plugins that are tailored to their architecture. With this approach, tool builders are required to repeat the implementation effort for supporting a language in many development environments.

The language server protocol (LSP) aims to reduce the implementation effort by standardizing the interprocess communication between a development environment (the language client) and a tool (the language server) that can provide language-semantics-aware features, such as autocompletion or documentation on hover.[5] This approach allows tool builders to implement the aforementioned features once by not tailoring to a specific development environment's architecture.

The LSP is based on JSON-RPC, a stateless, lightweight remote procedure call (RPC) protocol.[6] In JSON-RPC, clients make `Requests` to servers, which are expected to answer with a `Response`. A `Notification` is a special kind of `Request` that does not expect a `Response` from the server.

Consider a static analysis tool that provides type information for method parameters in a given programming language. Figure 3.1 illustrates an example sequence of LSP interactions if the tool were to be implemented as an LSP language server.

[5] `https://microsoft.github.io/language-server-protocol/overview`
[6] `https://www.jsonrpc.org/specification`

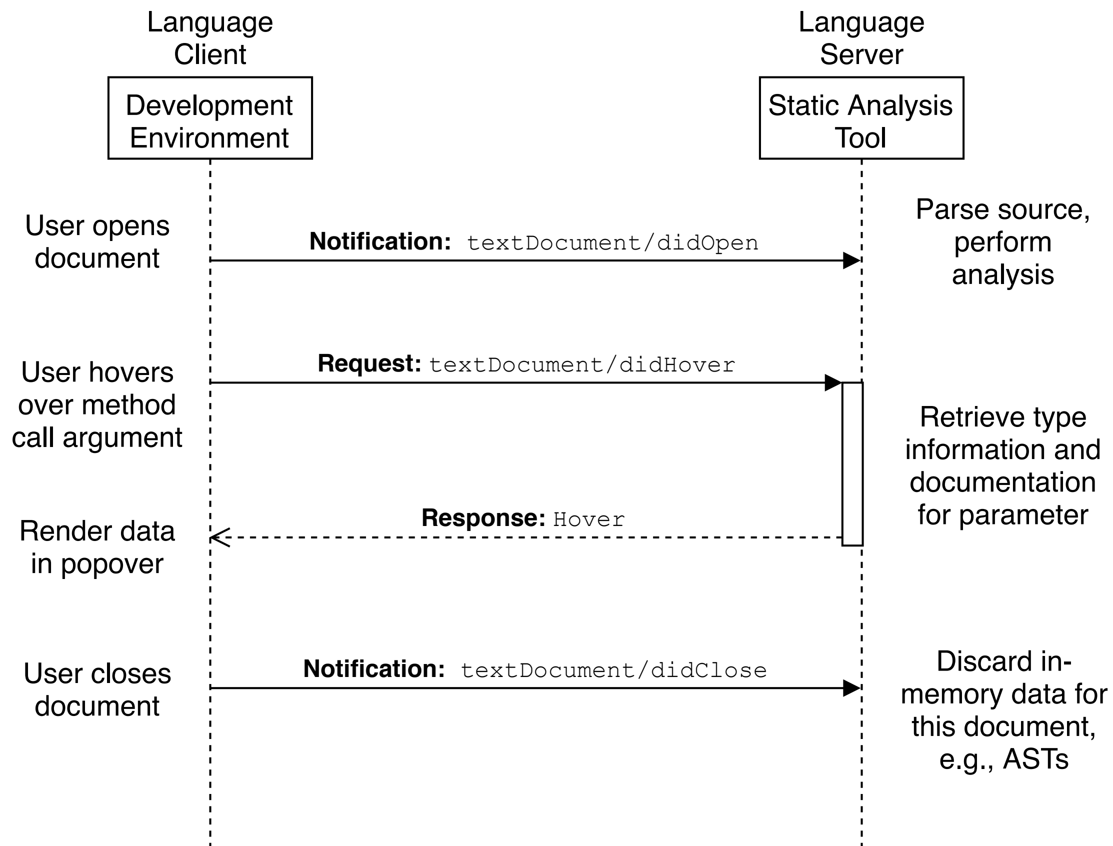


Figure 3.1: Example LSP communication when user overs hover method call argument

The following list elaborates on the interactions in Figure 3.1:

1. The user opens a source file in their development environment. The development environment sends a `DidOpenTextDocument Notification`[7] that contains parameters identifying the source file.
2. The static analysis tool parses the relevant source files and performs semantic analysis, *e.g.,* builds ASTs and populates symbol tables and keeps them in-memory for faster querying in subsequent requests.
3. The user now hovers over a particular method call argument, triggering a `Hover Request`[8] by the development environment with parameters identifying the source location[9].
4. The tool uses the source location information from the `Hover Request` parameters and the semantic information extracted in step 2 to determine that the user is hovering over a method call argument.

[7] https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_didOpen
[8] https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_hover
[9] https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textdocumentpositionparams

5. The tool resolves the argument type and answers to the client with a `Hover Response`[10] containing a string representation of the argument type. This particular development environment now displays the string representation in a popover window at the hover location.

6. The user eventually closes this source file, triggering a `DidCloseTextDocument Notification`.[11] The tool discards any redundant in-memory data structures to preserve memory.

A prerequisite for the LSP approach is that the development environment can act as a language client, *i.e.,* it can map user interactions to LSP `Requests` to a language server. This prerequisite is typically satisfied by providing a development environment plugin that sets up communication with one specific language server. An example implementation of this approach can be seen in in vscode-solargraph[12] (a Visual Studio Code plugin that communicates with a Ruby language server). Alternatively, a plugin can be configured to communicate with various language servers, as seen in ale[13] (a general purpose Vim plugin for linting and communicating with various language servers). Many language client implementations take care of starting up the language server as a subprocess as soon as they are initialized, *e.g.,* when the development environment starts up or a relevant source file is opened. The communication traditionally occurs via standard in/out or a TCP socket.

In summary, the LSP specifies the interactions between a development environment and the implementation of common language-semantics-aware features, so that the implementation can be reused by any environment that speaks the LSP. In section 5.2, we demonstrate how AUDREY implements an LSP language server in order to support many IDEs.

[10] `https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_hover`
[11] `https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_didClose`
[12] `https://github.com/castwide/vscode-solargraph`
[13] `https://github.com/w0rp/ale`

4
Audrey in a Nutshell

In this chapter we present the components found in AUDREY and how they interact together. Figure 4.1 shows an overview of the three major components in our current implementation:

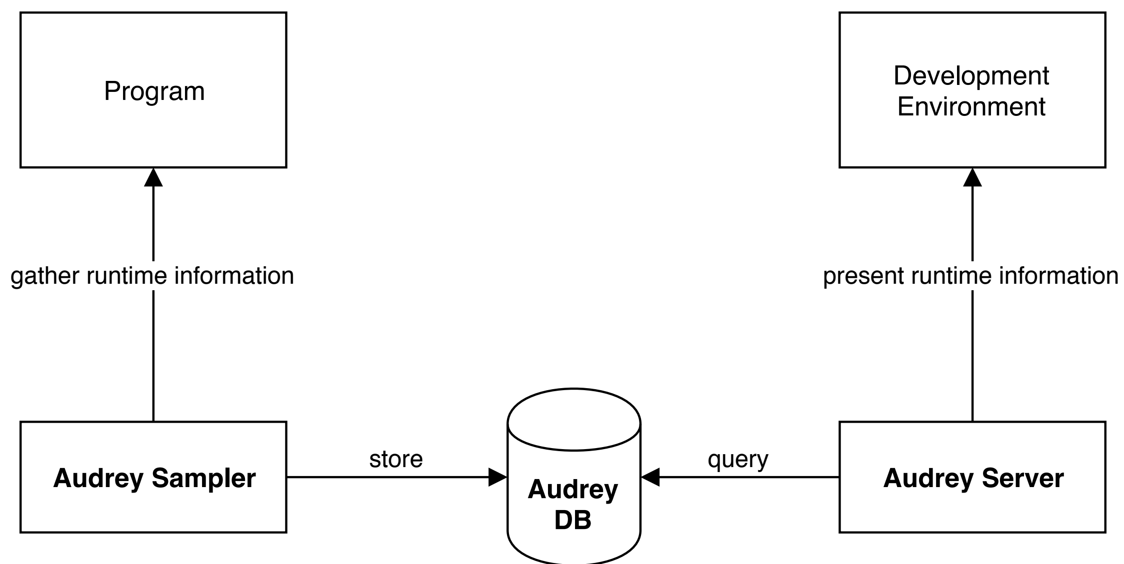


Figure 4.1: Overview of AUDREY components and their relationships

- AUDREY *Sampler*: a GraalVM tool that collects runtime information using the Truffle instrumentation API and stores it in a database.
- AUDREY *DB*: a Redis database that holds the collected runtime information in a language-agnostic format.
- AUDREY *Server*: an LSP language server that queries the database and exposes the information to development environments

4.1 Design Rationale

Our design choices were primarily driven by the requirement of a language-agnostic implementation in order to reuse the work for as many languages as possible. We chose Truffle in that regard for AUDREY Sampler for the following reasons:

- We gather runtime information for various different languages by implementing one tool in Java.
- We operate on a high-level structured intermediate representation, *i.e.,* a guest language AST, which is somewhat close to the source code *and* contains enough semantics to extract useful runtime information.
- We control the instrumentation at run time, *e.g.,* turn it off, schedule it, *etc.*
- There exist Truffle implementations of highly dynamic languages, such as Ruby, R or JavaScript, that typically suffer from a lack of static analysis tools.

AUDREY DB is a Redis database that holds a set of JSON serialized samples. This component is easily interchangeable and the primary motivation for choosing Redis as the initial database backend was its small resource usage and availability of client libraries for various languages. AUDREY Sampler uses Lettuce[1], a Java Redis client that allows us to asynchronously store samples.

Implementing AUDREY Server as an LSP language server has the benefit of providing common features such as documentation on hover and autocompletion to as many development environments as possible.

To collect runtime information for the function `blend` in Listing 1.1, the developer can simply run any piece of code that invokes this function with AUDREY Sampler enabled, *e.g.,* by rendering a scene with the raytracer, running any tests that exercise the function or even running the raytracer in the actual production environment. By default this will result in AUDREY DB containing all encountered values for the parameters `c1`, `c2`, `w` and the return value.

When the developer hovers over the function declaration, the language client in the developer's development environment sends a `Hover Request` to AUDREY Server, which will query AUDREY DB to build a useful `Hover Response`, *e.g.,* the contents of Figure 1.1.

This is AUDREY in a nutshell. It comes with a sampler that allows developers to gather runtime information in any environment, *e.g.,* as part of the test suite in continuous integration or the actual production system. The sampler leverages the Truffle API in order to instrument many languages with one approach. The gathered information is stored in a language agnostic format in a Redis database that can be shared among developers and an LSP language server exposes the information to many development environments.

[1] `https://github.com/lettuce-io/lettuce-core`

5

AUDREY Implementation

In this chapter we elaborate on how we use the Truffle API and the LSP to keep AUDREY language- and IDE-agnostic.

5.1 AUDREY Sampler

We leverage the Truffle framework to build AUDREY Sampler, the part of our system that gathers runtime information for multiple languages and stores it for later querying. We illustrate how AUDREY Sampler communicates with the Truffle API to collect all the information needed to provide runtime *examples* to development environments later on. We use Ruby and JavaScript to showcase the behavior for two different languages.

Our current implementation collects a simple textual representation of any Truffle language construct passed in as arguments to or returned from AST nodes tagged with *root*, which typically marks roots of functions, methods or closures. Additionally, some metadata is collected, such as type and source location information and identifiers for more precise source location mapping.

Listing 5.1 and Listing 5.2 show a Ruby and JavaScript snippet defining and calling a simple `add` procedure with various types. Both of these snippets will result in two examples being collected for the parameter `a` when executed with our tool enabled. Listing 5.3 shows the JSON[1] representation of one example after AUDREY has performed the extraction. Executing the JavaScript example in Listing 5.2 would yield an example with the exact same structure, differences being the values for the textual representation and meta information, which of course varies from language to language.

[1] `https://www.json.org`

Listing 5.1: A simple add method in Ruby

```
1 def add(a, b)
2   a + b
3 end
4
5 add(1, 2)
6 add(["hi"], ["there"])
```

Listing 5.2: A simple add function in JavaScript

```
1 function add(a, b) {
2   return a + b;
3 }
4
5 add(1, 2);
6 add(["hi"], ["there"]);
```

Listing 5.3: JSON representation of one example for parameter a extracted from Listing 5.1

```
1  {
2    "value": "[\"hi\"]",
3    "identifier": "a",
4    "identifierIndex": 0,
5    "metaObject": "Array",
6    "rootNodeId": "Object#add",
7    "category": "ARGUMENT",
8    "source": "file:///path/to/example.rb",
9    "sourceLine": 1,
10   "sourceIndex": 10,
11   "sourceLength": 1
12 }
```

The essential components of a runtime example as seen in Listing 5.3 can be summarized as following:

- A textual representation of the value (parameter or return value) with a language specific formatting, *e.g.,* `"1"` or `"{red: 0.15, green: 0.72959, blue: 0.72959}"`
- A textual representation of the identifiers if they exist, *e.g.,* `"a"` for the parameter name
- The meta-object of the extracted value. A Truffle meta-object reveals a value's kind and its features, *e.g.,* in the form of a language-specific class or interface representation. For example, asking TruffleRuby for the meta-object typically yields a Ruby class, *e.g.,* `"Integer"`, whereas asking GraalJS will yield a JavaScript type or a constructor function, *e.g.,* `"number"`.
- The root node identifier, i.e. method identifier in Ruby or function name in JavaScript, *e.g.,* `"Object#add"`
- Source location information, e.g. a URI to the instrumented source file with line and column information

We extract these components by injecting a custom AST node into the Truffle language ASTs. Figure 5.1 visualizes a simplified AST of the `add` function in Listing 5.2 with AUDREY Sampler enabled. For

the use case of extracting parameter and return value examples, we inject one AUDREY Sampler node right after Truffle AST nodes tagged with *root*, *e.g.,* the function node `add` in Listing 5.2, and gather parameters and return values.

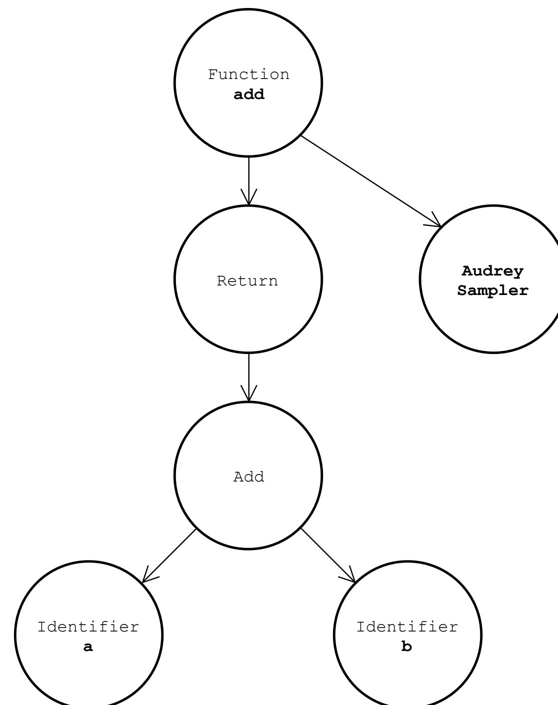


Figure 5.1: Simplified AST of the `add` function in Listing 5.2 with a custom AUDREY Sampler node that collects run-time examples

The Truffle instrumentation framework notifies our node about two events:

- `onEnter`: when the root node starts executing, e.g. when we enter a method in Ruby or a function in JavaScript
- `onExit`: when the root node has finished executing successfully, e.g. when we return from a method in Ruby or a function in JavaScript

On the `onEnter` event, we extract the examples for the parameters that are being passed in. On the `onExit` event, we extract the examples for potential return values. Note that in the following descriptions, we give a superficial overview of the communication with the Truffle API. At the time of writing the relevant event handler implementations can be found in the the `RootOnlySamplerNode` class of the AUDREY source code.[2]

The key to acquiring all example components are Truffle's interoperability capabilities. Truffle comes with a language-agnostic mechanism for efficiently calling foreign-language-specific operations [9]. We leverage this feature to delegate certain tasks to the Truffle languages themselves, such as acquiring a textual representation or language-specific meta information.

We only extract a textual representation for parameter and return values. While a short textual representation excludes many visualization possibilities, it is compact and familiar. It integrates well into

[2]`https://github.com/rathrio/audrey/`

any development environment and can be easily serialized and deserialized in any modern programming language [21]. We extract this textual representation using Truffle's cross-language interoperability. At the time of writing, at least TruffleRuby and the GraalVM JavaScript engine implement a `toString`[3] method for an arbitrary language-specific value. This representation is already serving as the string representation used in the GraalVM debugger using the Chrome DevTools Protocol.[4] In a similar manner, we extract the additional meta information, such as the identifier names and meta-objects. Listing 5.4 shows a simplified implementation illustrating how our approach uses the Truffle API to acquire the example components. Note that various implementation details are deliberately omitted.

Listing 5.4: An `onEnter` handler that extracts parameter samples using various Truffle API calls

```
1  void handleOnEnter(MaterializedFrame frame) {
2    Scope scope = env
3        .findLocalScopes(instrumentedNode, frame)
4        .iterator().next();
5
6    TruffleObject arguments
7        = (TruffleObject) scope.getArguments();
8
9    int keySize = getSize(keys);
10
11   for (int index = 0; index < keySize; index++) {
12     String identifier = (String) read(keys, index);
13     Object valueObject = read(arguments, identifier);
14     Object metaObject = getMetaObject(valueObject);
15
16     Sample sample = new Sample(
17        identifier,
18        index,
19        getString(valueObject),
20        getString(metaObject),
21        "ARGUMENT",
22        sourceSection,
23        rootNodeId,
24     );
25
26     storage.add(sample);
27   }
28 }
```

On lines 2 to 4, we look up the scope for the node that we are currently instrumenting. When instrumenting a Ruby method for instance, the relevant scope here would be the local scope within the method body. This scope is useful for retrieving the method arguments, as illustrated on lines 6 and 7. For example, the call on line 5 in Listing 5.1 results in an `onEnter` event for the method `add` where arguments to the scope would contain a `TruffleObject` representing the Ruby integers `1` and `2`.

A `TruffleObject` represents a guest language entity that can be interacted with using Truffle's interoperability capabilities. A first example is seen on line 9, where we use these capabilities in the `getSize` call to determine the number of arguments. More examples can be found in the loop starting on line 11. The `read` call on lines 12 and 13 ask the guest language to retrieve the identifier and values of

[3] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/TruffleLanguage.html#toString-C-java.lang.Object-`

[4] `https://www.graalvm.org/docs/reference-manual/tools/#debugger`

the arguments. The `getMetaObject` call on line 14 similarly retrieves the meta-object of an argument, *e.g.,* an object representing a Ruby class. The `getString` calls on line 19 and 20 delegate to the guest language to retrieve the aforementioned textual representations as seen in Listing 5.3.

The `storage.add(sample)` call in on line 26 stores the extracted sample in a configurable storage backend. By default, our system stores a JSON representation of the example into a Redis[5] database.

Truffle and its guest language implementations are still in their early development stages. Consequently, one has to deal with some inconsistencies within the Truffle languages themselves. We elaborate on this in section 6.1. For the use case of AUDREY, the benefits of Truffle as mentioned in chapter 4 outweigh these issues.

Communicating with the guest languages comes at a large performance cost. Depending on the application profile, the instrumentation overhead can bring the application to a near halt, which is not feasible when one wants to employ AUDREY Sampler in a production or other performance critical environment. We mitigate this issue by exposing various filtering and sampling strategies, so that it can be tailored to the developer's needs. This is illustrated in section 6.3 on a JavaScript raytracer.

Any mechanism that is logging arbitrary data of a running system is prone to leaking sensitive data and AUDREY Sampler is no exception. When AUDREY is utilized in an uncontrolled manner, one might store unencrypted passwords or credit card information in the Redis DB. We discuss this in section 6.4 and chapter 7.

With this part of the system in place, developers can collect run-time examples for any running application implemented in a Truffle language. We achieve this by implementing a tool using the Truffle instrumentation API and provide filtering and sampling options to control the instrumentation overhead. The extracted examples are serialized in a language-agnostic format into a Redis database, so that they can be served to development environments using the LSP.

5.2 AUDREY Server

AUDREY Server is implemented as an LSP language server in order to expose the runtime information in an IDE-agnostic way. The main difference to traditional language servers is that AUDREY Server supports requests for many languages as opposed to one. For instance, it does not matter whether the user hovers over a Ruby method or a JavaScript function. Our implementation of AUDREY Server handles the `Hover Request` and `Response` in a language-agnostic manner, while pluggable language-specific backends implement the semantics needed to retrieve the relevant runtime information for the requested source location.

Our current implementation of AUDREY Server is a standalone language server that comes with pluggable language-specific backends. AUDREY Server handles the LSP requests by clients, but delegates to a language-specific backend for semantic queries. The language-specific backend has access to a parser implementation of choice, *e.g.,* one that is error tolerant and annotates the resulting ASTs with source location information for tooling purposes.

Figure 5.2 illustrates how each component from the development environment to the AUDREY DB interact with each other to provide examples for multiple languages in our current architecture:

1. The user opens a Ruby source file in their development environment. The development environment sends a `DidOpenTextDocument Notification` to AUDREY Server.
2. AUDREY Server recognizes that the client opened a Ruby file and thus informs the Ruby backend that the file was opened.

[5] `https://redis.io`

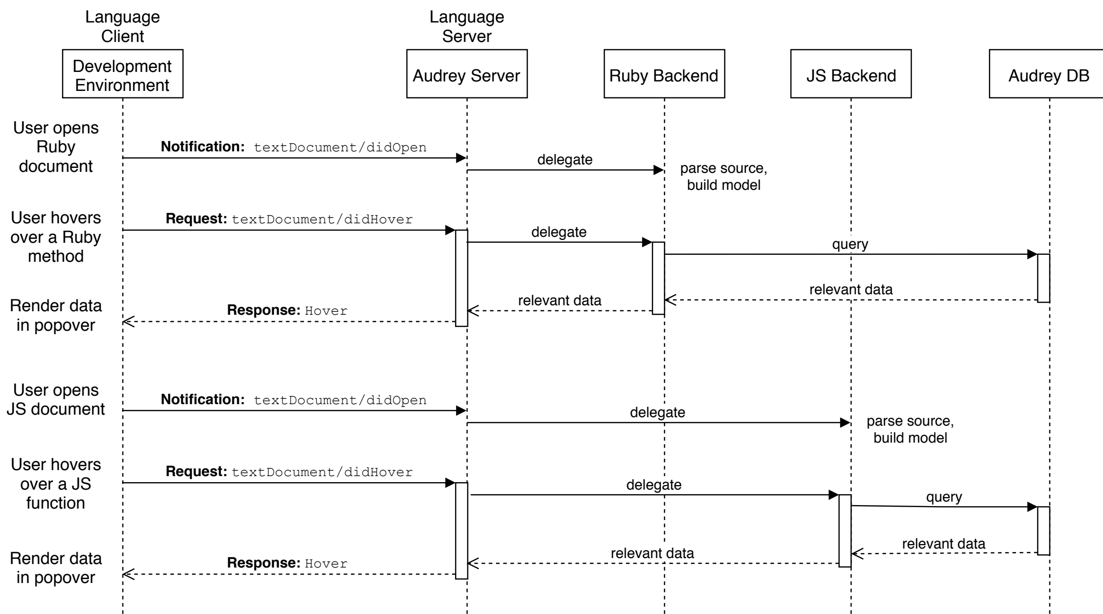


Figure 5.2: Communication with AUDREY Server and its language-specific backends

3. The Ruby backend parses the relevant source files keeps ASTs in memory for faster querying in subsequent requests.
4. The user now hovers over a particular Ruby method, triggering a `Hover Request`.
5. AUDREY Server delegates the request to the Ruby backend, which can determine what exactly the user is hovering over as a result of having access to the previously parsed ASTs with source location information.
6. The Ruby backend can now precisely query the AUDREY DB, *e.g.,* "Retrieve all examples for the Truffle node `SomeClass#some_method`".
7. The retrieved data is processed and propagated back to AUDREY Server, which can now prepare the hover response for the development environment.
8. The process is repeated for a JavaScript source file, whereas now the JavaScript backend will perform the parsing and querying.

Ideally, all languages would expose a language-agnostic tooling API in the likes of clang[6], *i.e.,* we could reuse parts of the language front ends for tooling purposes as well. Our current approach enables us to easily integrate a language-agnostic Truffle tooling API backend in the future, while retaining the flexibility of falling back to a language-specific backend, should we require more language-specific semantics. For Truffle languages, a language-agnostic tooling API would render the need for language-specific backends obsolete.

We elaborate on this idea and the challenges of mapping potentially changing source code to the collected information in section 6.2.

[6] https://clang.llvm.org

6 Discussion

We started building AUDREY with three requirements in mind:

- AUDREY is language agnostic, i.e., it can be reused for many languages with little to no engineering effort.
- AUDREY is development environment agnostic, i.e. it can be reused for many editors and IDEs with little to no engineering effort.
- The runtime information can be gathered with minimal instrumentation overhead, so that it is practical to do so in a production environment.

In this chapter, we discuss the major challenges and limitations we have encountered throughout the process of fulfilling these requirements and how we dealt with them.

6.1 Finding an IR for language-agnostic instrumentation

For gathering runtime information language-agnostically, the challenge was finding an instrumentable IR that is similar enough in many languages, so that we can reuse one approach for all of them. Take Ruby and JavaScript for instance. Both implement functions in the form of a subroutine that may take some parameters and return a value. To extract parameters and return values in the default C implementation of Ruby, developers can utilize the built-in TracePoint API[1] in order to monitor call events in Ruby's bytecode VM, *i.e.,* they write Ruby code that is tailored to an implementation specific IR. The ECMAScript standard, the specification that JavaScript implementors ideally comply with, does not prescribe an API for instrumentation. Consequently developers build ad-hoc instrumentation tooling or rely on third-party solutions, such as Google's tracing framework.[2] In other words, they write JavaScript code that is tailored to the most commonly used JavaScript runtimes.

This disparity in runtimes and instrumentation approaches makes it difficult to build one tool that can be reused. We have thus explored runtimes that come with a high-level target IR suitable for multiple

[1] https://ruby-doc.org/core-2.6.1/TracePoint.html#class-TracePoint-label-Events
[2] https://github.com/google/tracing-framework

languages, such as the JVM and Microsoft's Common Language Runtime (CLR) and have landed at the Truffle AST, which became the common underlying infrastructure for information gathering for the reasons described in chapter 4:

- We gather runtime information for various different languages by implementing one tool in Java.
- We operate on a high-level structured intermediate representation, *i.e.,* a guest language AST, which is somewhat close to the source code *and* contains enough semantics to extract useful runtime information.
- We control the instrumentation at run time, *e.g.,* turn it off, schedule it, *etc.*
- There exist Truffle implementations of highly dynamic languages, such as Ruby, R or JavaScript, that typically suffer from a lack of static analysis tools.

Throughout the development of AUDREY Sampler we have however encountered difficulties that can be attributed to Truffle languages not being as battle-tested as JVM or CLR languages:

- There are little to no guides on how to set up, structure, test or ship a GraalVM tool. What worked best for us is to reference the implementation of an existing tool such as the *e.g.,* the profiler.[3]
- One has to become familiar with Truffle's optimization strategies which also lack documentation from a tool building perspective. For instance, one has to be aware when to explicitly mark methods as Truffle boundaries[4] or what actions will result in a deoptimization.
- The behavior when retrieving a string representation of a value is not consistent among the guest languages. GraalJS abbreviates the string representation of arbitrarily large objects, *e.g.,* by substituting deeply nested values with `"..."`: `{ person: { name: "Bob", age: 42, address: ... } }`. TruffleRuby outputs everything, which is a problem when storing examples for larger data structures, such as Arrays with thousands of elements. GraalPython seems to simply output the default `__str__()` call result, which is not useful for arbitrary values.

In conclusion, we believe that the Truffle AST is currently the best answer to "What is a practical underlying run-time representation of a running program that allows us to gather the information for as many languages as possible?".

6.2 The missing tooling infrastructure

While the LSP provides a pragmatic answer to "How do we expose the information to various development environments without writing multiple IDE-specific plugins?", the communication with development environments emerged to be the simpler challenge when compared to supporting multiple languages without additional engineering effort.

The main challenge of presenting runtime information for multiple languages is the lack of a universal language-agnostic front end. One approach is to simply ignore any language semantics and just work with source location information, *e.g.,* when the user is hovering on line 5, simply present all data collected for line 5. This approach works as long as the source code is exactly the one that was instrumented and the developer does not perform any modifications. We believe this approach is not practical, since we

[3] https://github.com/oracle/graal/tree/7017a81d23116105023c48e2be89355f79cbe1d6/tools/src/com.oracle.truffle.tools.profiler/src/com/oracle/truffle/tools/profiler

[4] https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/CompilerDirectives.TruffleBoundary.html

want to provide the runtime information for a function even if the source code was modified within reason. Additionally, the instrumented source code may not be the one that the developers work on, which is the case for many JavaScript projects that use transpilers to utilize modern JavaScript features the runtime does not support yet natively.

Ideally, Truffle languages would expose parts of their front ends over a standardized "front end" API in order to simplify semantic source code queries. For example, when a user hovers over a location in a Ruby source file, AUDREY Server could communicate with TruffleRuby to precisely determine where the user is hovering, *e.g.,* over a parameter of a Truffle root node called so and so. Stolpe *et al.* are working on enhancing the Truffle API in order to improve language server support[5], but it did not satisfy our language-agnostic needs at the time of writing.

A difficulty with reusing the Truffle language front ends is retrofitting their design to development tooling. By default, language front ends are optimized for building up the intermediate representations so that they are useful for the language backend. In the case of Truffle languages, their front ends are optimized for building up a Truffle AST. However, front ends powering a language server need to be designed in a way to efficiently answer various semantic queries, such as "Where is the definition of this method?" (for a `Goto Definition Request`[6]) or "What methods does this local variable respond to?" (for a `Completion Request`[7]) while dealing with potentially syntactically invalid source code. We believe that Truffle has the potential to become the dynamic-language counterpart to what clang has become for the C language family with respect to tool development.

An alternative approach that leverages existing language servers is to implement AUDREY Server as a lightweight service that can be integrated into any language-specific language server that already uses a front end optimized for tooling purposes. This way AUDREY Server can expect more precise requests, *e.g.,* instead of an LSP `Hover` request with plain source location information, it could expect a request containing semantic information, such as a Ruby method name, eliminating the need for AUDREY Server to communicate with a language-specific front end. As a proof-of-concept for this approach, we implemented AUDREY Server as an HTTP service, and we enhanced solargraph[8], a Ruby language server, so that when solargraph receives an LSP `Hover Request`, it also fetches runtime information from AUDREY using the semantic information it has already extracted. This approach is straightforward to implement, but it requires implementation effort in various third-party language servers.

6.3 Instrumentation overhead

Additional behavior comes at a performance cost. This is especially the case with AUDREY Sampler, which by default executes additional code for every call target invocation. We mitigate this by providing the developer with various sampling options.

Figure 6.1a shows the performance profile of Burmister's raytracer[9] when AUDREY Sampler collects information for every single JavaScript function invocation:

What is presented here is the time it takes the raytracer to repeatedly render a scene. Without any instrumentation by AUDREY Sampler, GraalJS rapidly reaches peak performance at < 5ms render time per scene, whereas when AUDREY Sampler is gathering everything, we observe a near thousandfold increase of render time.

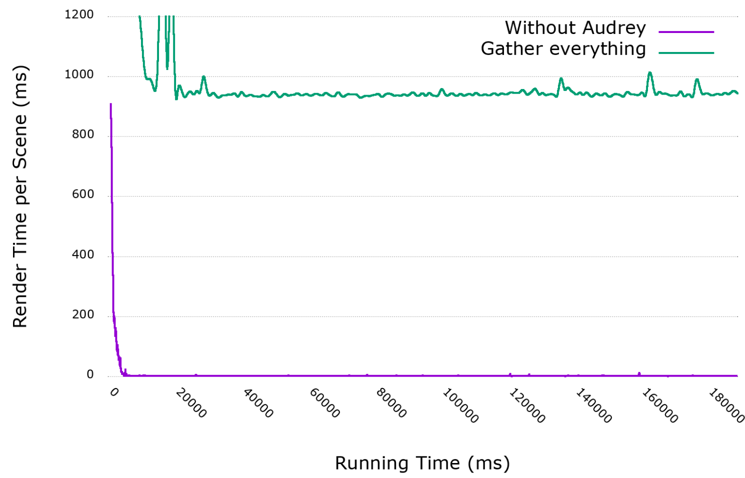
[5] `https://github.com/oracle/graal/pull/764`

[6] `https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_definition`

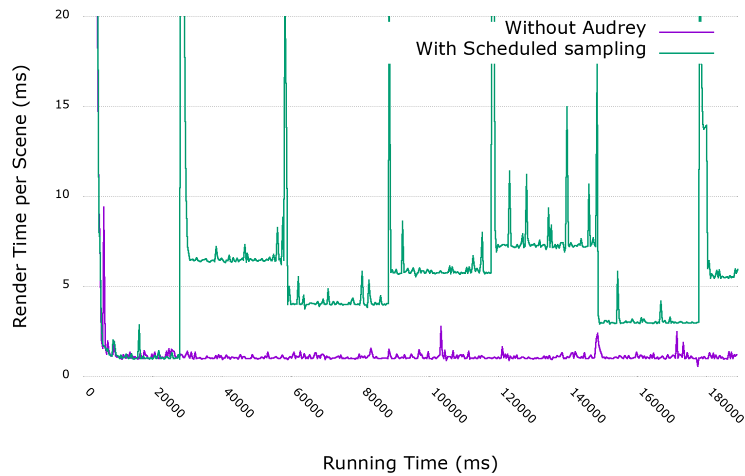
[7] `https://microsoft.github.io/language-server-protocol/specifications/specification-3-14#textDocument_completion`

[8] `https://github.com/castwide/solargraph`

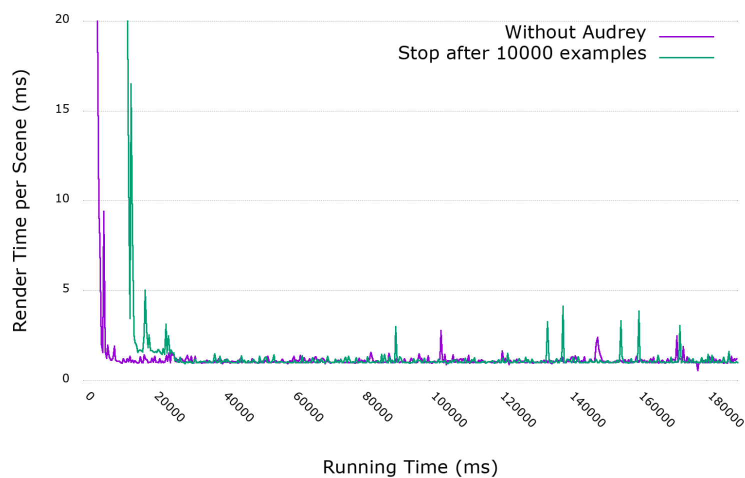
[9] `https://chromium.googlesource.com/v8/v8/+/4.3.18/benchmarks/raytrace.js`



(a) Profile when gathering at every function invocation, *i.e.,* AUDREY samples all parameters and return values for every single function call by injecting a sampler node at every encountered root node



(b) Profile with scheduled sampling, *i.e.,* AUDREY instruments separate parts of the application over time by injecting or removing sampler nodes at certain source locations



(c) Profile when disabling instrumentation after 10000 samples, *i.e.,* AUDREY removes all sampler nodes after the desired amount of samples have been collected, resulting in the raytracer eventually reaching peak performance

Figure 6.1: Performance profile of a JavaScript raytracer with different AUDREY sampling strategies

This behavior is due to the fact that the AST nodes inserted by AUDREY Sampler are more challenging to optimize by the Truffle framework. This makes gathering runtime information for all invocations not practical for most medium to large programs.

AUDREY mitigates the overhead by providing various sampling options. In practice, a developer might not want to collect data for every single function. Thankfully, the Truffle instrumentation API allows nodes to be inserted when the source section matches certain criteria, *e.g.,* when the file path matches a certain substring. AUDREY allows developers to provide patterns for source sections filters via a command line interface in order to more precisely control what exactly is instrumented.

We also allow developers to control when and how often AUDREY gathers the information. This is especially useful for long running programs, where AUDREY can be configured to sample very infrequently, allowing the Truffle framework optimizations to take place and deoptimizing when an AST node needs be inserted for sampling.

A more automated option is scheduled sampling. When scheduled sampling is activated, AUDREY will distribute the encountered functions into multiple groups and sample them at different times, resulting in the performance profile illustrated in Figure 6.1b.

The render time spikes are a result of Truffle having to deoptimize the affected portions of the program where AST modifications take place. The interval and number of groups are parameterizable as well, since the optimal configuration is very program specific. Ideally, the scheduler would incorporate a prerecorded application profile of a characteristic run to further minimize the instrumentation overhead, which is another open problem and requires more research.

The best option when it comes to performance is not instrumenting at all, *e.g.,* by stopping instrumentation at run time when a satisfactory number of examples have been collected. The act of stopping instrumentation will again result in a brief performance penalty due to the deoptimization step required when removing AST nodes, but allows a program to eventually reach peak performance. Figure 6.1c shows the performance profile of the raytracer where render time is impaired at first, but it eventually reaches peak performance, since AUDREY can remove the instrumenting AST nodes at run time after they have collected 10000 samples.

While this option is beneficial for long time performance, it might not be ideal for the diversity of the collected samples. For instance, the application and flowing data profile may change after 10000 samples, which AUDREY would disregard with this sampling strategy. Detecting changes in the data profile and reenabling data gathering is one of the directions discussed in chapter 7.

The aforementioned approaches are all combinable and configurable. We believe that there is no silver bullet when it comes to mitigating the performance overhead for an arbitrary program. Therefore the best approach is to have AUDREY expose all sampling strategies to the developer. In other words, the answer to "How can we keep the instrumentation overhead to a minimum while gathering the information?" to a large degree depends on the application profile and we do not have a universal solution to keep the instrumentation overhead to a guaranteed minimum with the Truffle instrumentation framework.

6.4 Information leakage

At run time, a program may work with various confidential data. Consider the Ruby `create_user` method in Listing 6.1 that takes a user name and a password and creates a new user with an encrypted version of the password.

Listing 6.1: A Ruby method that works with confidential data

```
1 def create_user(username, password)
2   database.create_user(username, encrypt(password))
3 end
```

When AUDREY Sampler is allowed to instrument this Ruby method, AUDREY DB will contain examples for the parameters `username` and `password`, where the `password` parameter seems to be unencrypted.

Unfortunately, it is not deducible from the built in language constructs which parameters contain sensitive data that should not be leaked in this fashion. Therefore we delegate this task to the developer by exposing source filtering options. This approach is not satisfactory as it is fairly error prone, *e.g.,* the developer not only has to disable instrumentation for the code in Listing 6.1, but also at all call sites of `create_user`.

An approach that could be more robust for AUDREY Sampler is if the data itself were to be marked as sensitive so that we can prevent sampling. This could be achieved with custom data structures that prohibit certain operations such as outputting a string representation to standard output.

In conclusion, the most pragmatic answer with state of the art methods to "How do we prevent gathering sensitive information?" is to expose filtering options to the developer. Identifying security regions that should not be instrumentable is an open problem in general and requires further research [5].

7
Future Work

AUDREY Server currently only responds to an LSP `onHover` request. Hence we only provide improved textual documentation when hovering over certain locations. However, the LSP standardizes many more developer requests, such as go-to-definition, autocompletion or diagnostics (error messages on semantic analysis). One could utilize the runtime information in AUDREY DB to provide more precise answers to these requests, *e.g.,* by improving type inference and type checking precision based on the encountered run-time types.

Most IDEs that implement LSP clients are very limited in what they allow to be rendered in a hover response, *i.e.,* most only support textual contents with limited markdown support. Should they eventually support rendering of a custom web view, the visualization possibilities would open up drastically. One could for instance easily render interactive plots for numerical method parameters with d3js[1] with a textual "fallback" representation for clients that do not support web views.

As discussed in section 6.2, source code is prone to change. This requires specialized language front ends with error tolerant parsers to provide tooling while the developer is editing code. A step in this direction is to investigate how feasible a front end API for tooling purposes is for Truffle and whether it is desired by the Truffle authors.

We provide outdated information if it can still be mapped to the source code. For instance, if we have some information for a Ruby method `Database#create_user`, we will display the information even if the method is encountered at a location different than the one that was originally instrumented, *e.g.,* after some other piece of code has been inserted above it during development. However, if up-to-date runtime information could be gathered within reasonable time, changes made to the source could should ideally update the database. This could for example be achieved by reacting to the LSP `DidChangeTextDocument Notification`. AUDREY Server should allow to be configured to run a small test suite with good coverage to repopulate the Redis database with up-to-date examples.

The collected examples could profit form additional meta information, such as a unique trace identifier. This way, we could more precisely visualize which method arguments resulted in which return values. Having trace information could help in supporting backward-in-time debugging [10] should the IDE provide the infrastructure.

AUDREY Sampler instruments source code based on source location information, *e.g.,* the developer

[1] `https://d3js.org/`

can control which folder or file is instrumented. In order to avoid leaking sensitive data as illustrated in section 6.4, it could be beneficial to disable instrumentation on a per language construct basis. For instance, if the developer declares that the information passed to a specific function should not be gathered, AUDREY Sampler could employ taint checking to disable sampling of this information in the whole program.

When AUDREY Sampler is configured to stop instrumentation after it has collected a certain number of examples, the examples might be too similar depending on the program profile. Truffle has a mechanism in place when the compiled code encounters data that it is not specialized for, *e.g.,* when a method is suddenly called with new data types. In that case, Truffle will initiate a deoptiomization from the compiled code to the AST interpreter. AUDREY Sampler could reenable instrumentation for code that was deoptimized by Truffle which could result in more diverse examples.

AUDREY has not been tested in a real world scenario and would thus profit from various case studies. As AUDREY is highly configurable, it would especially be valuable to see how many and which examples are useful to support developer needs. It is also unclear which sampling strategies would work best for larger applications. Case studies would help evaluating which options make sense and which configurations of AUDREY would make sensible defaults for common application profiles.

8 Conclusions

Many dynamically-typed languages share the same issues during program comprehension and software evolution tasks due to the lack of static type information and consequent shortage of static analysis tools. In this thesis, we explore a novel strategy that gathers runtime information for multiple languages with controllable overhead and is capable of exposing the gathered information to multiple development environments. The main contribution of this thesis is a discussion of the challenges and limitations of state of the art technologies when it comes to implementing such a strategy with the following key requirements:

- The strategy is language agnostic, i.e., it can be reused for many languages with little to no engineering effort.
- The strategy is development environment agnostic, i.e. it can be reused for many editors and IDEs with little to no engineering effort.
- The runtime information can be gathered with minimal instrumentation overhead, so that it is practical to do so in a production environment.

We present AUDREY, a system that gathers runtime examples of method parameters and return values for multiple languages and exposes the information to many development environments. The runtime information gathered by AUDREY tries to not only compensate for the lack of static type information, but enhance program comprehension by providing textual representations of example parameter and return values, which is also beneficial to statically typed languages. AUDREY consists of three interchangeable components: a sampler that gathers the runtime examples, a database that holds the examples in a language-agnostic format and a server that exposes the examples to development environments.

The aforementioned idealistic requirements entail various research questions that explore the limits of current technologies, which we discuss with the help of AUDREY as a practical real-world example that is expectedly not truly language and IDE-agnostic. We conclusively answer the research questions posed in chapter 1 as follows:

What is a practical underlying run-time representation of a running program that allows us to gather the information for as many languages as possible?

Gathering runtime information for all languages would require a universal language runtime. Thus, the main challenge of gathering information for multiple languages is finding a shared intermediate representation that is instrumentable. Our search resulted in the Truffle language implementation framework for the GraalVM, as it allows us to write tool in Java that can instrument multiple guest language ASTs. We illustrate how AUDREY is implemented using the built-in instrumentation capabilities of Truffle in section 5.1 and elaborate on the challenges of working with Truffle in section 6.1.

How can we keep the instrumentation overhead to a minimum while gathering the information?

With the Truffle instrumentation framework, we cannot reliably guarantee a specific minimum overhead, as the instrumentation overhead depends heavily on the application profile. We demonstrate this on a raytracer example, which is particularly affected due to many function invocations. When gathering examples at all function invocations, the performance overhead is near thousandfold. We however drastically reduce the overhead by providing various sampling options, such as a scheduled sampling strategy, which allows developers to configure AUDREY to sample different source code locations over time. This works especially well with the Truffle instrumentation framework, as the applications may recover peak performance when the instrumentation is dynamically disabled.

How do we expose the information to various development environments without writing multiple IDE-specific plugins?

Instead of tailoring to a specific plugin architecture of an IDE, AUDREY remains IDE-agnostic by speaking the LSP. We discuss our attempt at building a language-agnostic LSP language server. What currently prevents a truly language-agnostic language server is the lack of a tooling focused language front-end in the likes of clang for all Truffle languages. Our compromise is having a language-agnostic server delegate requests to interchangeable language-specific backends. We however believe that a language agnostic front-end for tooling purposes is a sensible and natural goal for Truffle languages that is currently missing.

How do we retrieve the relevant runtime information for a given source location in the development environment?

As for the previous question, we solve this by hand-crafting language specific backends for the LSP server that depend on a language specific parser, so the requested source location can be more precisely mapped to a node in the specific language AST. The major limitation here is again the lack of a language agnostic front-end, which would reduce the amount of language specific engineering effort.

How do we keep the stored information up to date when the source code is modified?

Gathering runtime information is very specific to an application. Some applications may have a comprehensive test suite that would result in a satisfiable set of gathered examples when rerun, while others have to rely on production data that cannot easily be regathered upon source code modifications. AUDREY currently does not provide an automated regathering solution. As a possible solution, we describe how one could utilize the LSP `DidChangeTextDocument Notification` to trigger regathering in chapter 7 for use-cases where regathering can be done within reasonable time.

Installing and Using Audrey

In this appendix we provide installation and usage instructions for Audrey so that it can be tested on a single machine.

A.1 Installation

A.1.1 Prerequisites

Before installing and running Audrey, ensure that the target machine has the following software installed.

- A UNIX based operating system such as Ubuntu[1] or macOS[2]
- Git[3]
- GraalVM Community or Enterprise Edition[4]
- Gradle[5]
- Redis[6]

Ensure that the environment variable JAVA_HOME is set to the Contents/Home subfolder in the downloaded GraalVM folder:

```
1 export JAVA_HOME=/path/to/graalvm/Contents/Home
```

Ensure that a redis server is running, for example by pinging the server with the CLI client. The client should output "pong" when the server is running with the default configurations:

[1]https://ubuntu.com/#download
[2]https://www.apple.com/macos/catalina/
[3]https://git-scm.com
[4]https://www.graalvm.org/downloads/
[5]https://gradle.org/install/
[6]https://redis.io

```
redis-cli ping
# => pong
```

A.1.2 Installing Audrey

Audrey can currently only be installed from source with the Gradle build tool.

Clone the repository[7] with git to a desired location on the local machine and change the working directory to the repository's root folder to proceed with the installation:

```
git clone git@github.com:rathrio/audrey.git /wherever/you/want/audrey
cd /wherever/you/want/audrey
```

At the time of writing, Audrey's sampler is a GraalVM tool that needs to be placed in the $JAVA_HOME/jre/tools/ subfolder so that GraalVM knows about its existence. Run the following gradle task to build a JAR and move it to the subfolder:

```
./gradlew install
```

On successful completion the Audrey JAR should be located at $JAVA_HOME/jre/tools/audrey. Verify the installation by running the following:

```
$JAVA_HOME/bin/js --jvm --help:tools | grep audrey
```

If the output contains descriptions for --audrey flags and switches, the installation was successful.

A.2 Usage

A.2.1 Gathering runtime information with Audrey

Gathering runtime information with Audrey is now a matter of running a program with one of the Truffle language executables found in the GraalVM bin subfolder and at least providing the --audrey switch to enable instrumentation with the default configurations.

To test this on a simple example, create a file add.rb with the following contents:

```
# in add.rb
def add(x, y)
  x + y
end

add(1, 2)
```

To store the example values 1 and 2 with some meta information in the redis database, run the following:

```
$JAVA_HOME/bin/ruby --jvm --audrey add.rb
```

Consult the output of --help:tools to get a list of various options.

[7] https://github.com/rathrio/audrey

A.2.2 Displaying runtime information in a development environment

The gathered examples can be displayed in any development environment with a language client that supports the LSP `Hover Request` and somehow renders the response.

At the time of writing, configuring such clients is not standardized and requires different configuration strategies depending on the development environment. In general, language clients communicate via a UNIX TCP socket or simply `STDIN` and `STDOUT` when the language server is spawned as a subprocess. For debugging purposes, we provide a gradle task that starts an AUDREY Server that listens to TCP port 8123:

```
1 ./gradlew startServer
```

Point a language client of choice to the TCP port 8123 and the communication should be set up. The `language_servers` subfolder contains various examples, such as `language_servers/lsp-sample`, a Visual Studio Code plugin that communicates with the debugging server started with `./gradlew startServer`.

Figure 3.1 shows a successfully rendered `Hover Response` in Visual Studio Code.

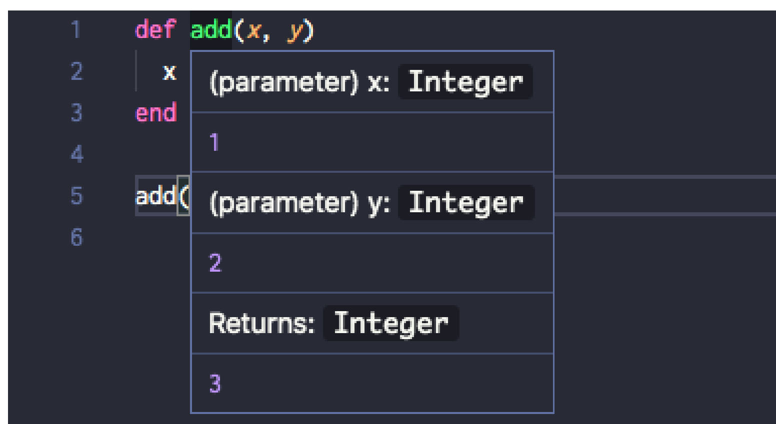


Figure A.1: Result of hovering over the `add` method in Visual Studio Code

Bibliography

[1] Ole Agesen. The Cartesian product algorithm. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag.

[2] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[3] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, August 2013.

[4] David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, pages 459–472. ACM, 2011.

[5] Cristina Cifuentes and Gavin M. Bierman. What is a secure programming language? In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA.*, pages 3:1–3:15, 2019.

[6] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *CoRR*, abs/1803.10201, 2018.

[7] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.

[8] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[9] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, 40(2):8:1–8:43, May 2018.

[10] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[11] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. *SIGPLAN Not.*, 47(10):683–702, October 2012.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] Nevena Milojković. Improving the precision of type inference algorithms with lightweight heuristics. In *SATToSE'17: Pre-Proceedings of the 10th International Seminar Series on Advanced Techniques & Tools for Software Evolution*, June 2017.

[14] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. Inferring types by mining class usage frequency from inline caches. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2016)*, pages 6:1–6:11, 2016.

[15] Nevena Milojković, Clément Béra, Mohammad Ghafari, and Oscar Nierstrasz. Mining inline cache data to order inferred types in dynamic languages. *Science of Computer Programming, Elsevier, Special Issue on Adv. Dynamic Languages*, 161:105–121, 2018.

[16] Morten Passow Odgaard. JavaScript type inference using dynamic analysis. Master's thesis, Aarhus University, 2014.

[17] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th Symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[18] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk. In *Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS)*, pages 76–77, 1998.

[19] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 47–65. Springer, 2007.

[20] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[21] Niko Schwarz, Mircea Lungu, and Oscar Nierstrasz. Seuss: Cleaning up class responsibilities with language-based dependency injection. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2011*, volume 33 of *LNCS*, pages 276–289. Springer-Verlag, 2011.

[22] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. Mining the ecosystem to improve type inference for dynamically typed languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '14, pages 133–142, New York, NY, USA, 2014. ACM.

[23] Boris Spasojević, Mircea Lungu, and Oscar Nierstrasz. A case study on type hints in method argument names in Pharo Smalltalk projects. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 283–292, March 2016.

[24] Michael Van De Vanter. Building debuggers and other tools: we can "have it all". pages 1–3, 07 2015.

[25] Hernán Wilkinson. Vm support for live typing: Automatic type annotation for dynamically typed languages. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, pages 9:1–9:3, New York, NY, USA, 2019. ACM.

[26] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM.