User-Defined Interface Mappings for the GraalVM

Alexander Riese
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
alexander.riese@student.hpi.uni-potsdam.de

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

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

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

ABSTRACT

To improve programming productivity, the right tools are crucial. This starts with the choice of the programming language, which often predetermines the libraries and frameworks one can use. Polyglot runtime environments, such as GraalVM, provide mechanisms for exchanging objects and sending messages across language boundaries, which allow developers to combine different languages, libraries, and frameworks with each other. However, polyglot application developers are obligated to properly use the right interfaces for accessing their data and objects from different languages.

To reduce the mental complexity for developers and let them focus on the business logic, we introduce user-defined interface mappings – an approach for adapting cross-language messages at run-time to match an expected interface. Thereby, the translation strategies are defined in an exchangeable and easy-to-edit configuration file. Thus, different stakeholders ranging from library and framework developers up to application developers can use and extend these mappings for their needs.

CCS CONCEPTS

• **Software and its engineering** → *Interoperability*; *Object oriented languages*; *Patterns*.

KEYWORDS

polyglot programming, language interoperability, portability, patterns

ACM Reference Format:
Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2020. User-Defined Interface Mappings for the GraalVM. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3397537.3399577

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3399577

1 INTRODUCTION

With the increasing complexity of software, it becomes all the more important to use the right development tools. By using suitable language constructs, frameworks, and libraries, the development time can be significantly reduced.

To use the most appropriate parts of each programming language for a given problem, polyglot programming provides a solution. Thereby, the developer can write code in multiple languages. To do so, polyglot runtime environments, such as GraalVM, provide mechanisms for language interoperability, such as access to objects across language borders.

In the case of GraalVM, the underlying framework provides interchangeability of primitive types and some selected basic data structures, such as lists. Since this mechanism is deeply integrated into the framework, it is rather hard to extend for other objects like dictionaries or even more domain-specific data types like data frames. This makes them cumbersome to use from different language contexts and therefore reduces the portability [3] of code.

To address this issue, we introduce user-defined interface mappings – an approach to provide cross-language interface mappings by generating suitable adapters. Thereby, application and library developers can extend the underlying interoperability mechanism to generate these adapters for their needs and can even share them with others.

Contributions. In this work, we introduce a concept to enable polyglot application developers to define and to use cross-language method mappings. Thereby, these predefined mappings can be used in a semi-automatic fashion by determining the expected interface from a method call and applying the corresponding mapping. To do so, the underlying polyglot environment is extended to make the internal message information accessible for tooling developers. In the course of the development, some questions and challenges are revealed, which will be discussed at the end.

Outline. In the next section, the required context for this work is given. Thereby, the underlying polyglot environment, GraalVM, and especially the interoperability mechanisms are explained. Building on this, the represented approach is outlined in the following

section. This is followed by a discussion about the trade-offs and challenges. Afterward, the approach is compared with other mechanisms providing access to data structures and object of different languages. Finally, the results are summarized, and we give an outlook for future work.

2 CONTEXT

This work focuses on language interoperability as provided by the GraalVM [13]. GraalVM is based on the Java HotSpot VM and provides interoperability between different programming languages. To archive this, language interpreters have to be implemented in Truffle, GraalVM's language implementation framework. Amongst others, implementations for Python[1], Ruby [7], and Squeak/Smalltalk [8] are provided.

Thereby, objects that can be shared between languages implement a common interface for interoperability. In this way, all languages have a shared understanding of message communication and messages can be sent to objects across language boundaries [6].

Moreover, GraalVM can be embedded into Java applications. For this, it provides proxy interfaces[2], ranging from array interfaces (e.g., `ProxyArray`) up to time zone interfaces (e.g., `ProxyTimeZone`). These interfaces can be implemented by using Java objects to mimic the behavior of specific objects in the target language. This way, it is possible to expose a domain-specific date object. For instance, a JavaScript `Date` object.

Additionally, GraalVM has support for *custom target type mappings*[3]. This mechanism allows the conversion of objects given a custom mapping. This functionality, however, is also limited to Java embeddings and thus cannot be used by developers of polyglot applications.

3 APPROACH

Our approach applies the adapter design pattern [5]. By wrapping an object from one language with an adapter, it can be exposed with appropriate interface from another language when called from there.

To build the adapter, the adaptee interface as well as the desired target interface is needed. While the former one can be determined by exploring the wrapped object itself, the latter is more critical. We assume that the target interface can be derived from the concept represented by the adaptee together with the language from which the adapter gets called. The underlying concept can be either expressed by the class of the wrapped object, or the user can explicitly define it, as shown in Listing 1.

Listing 1: Example interface to generate the first half of the adapter

```
from polyglot import PolyglotAdapter, mapping
adapter = PolyglotAdapter(dict(),
              mapping['Dictionary'])
```

To be able to share an understanding of concepts and their interfaces, these are defined as a language-agnostic representation. For the sake of simplicity, we describe the interface as a list of method names inside a YAML [2] file, as shown in Listing 2.

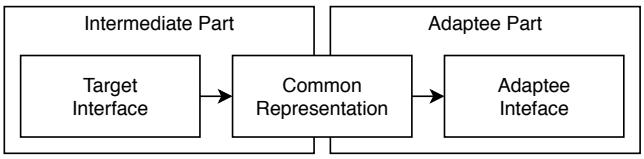

Figure 1: Illustration of the two-part adapter

Listing 2: Interface definition of Dictionary

```
identifier: Dictionary
methods:
  - lookup
  - add
  - includes
  ...
```

Given this interface definition, the method names are mapped in a YAML file, as shown in Listing 3. Thereby, each operation specified in the intermediate representation is assigned with a list of adaptee operations that implement the corresponding behavior. Whereas, it is not necessary to provide alias names for wrapped objects, since every translation should lead to the same result, it is crucial if the mapping is used in the other direction, as explained later. If the desired behavior is not provided from the adaptee, the format can be extended to define specific implementations on top of the shared operations.

Listing 3: Mapping file for Ruby hash and dictionary

```
lookup: "[]"
add: "[]="
includes:
  - include?
  - member?
...
```

By introducing a common representation of a shared interface, the implementation effort can be reduced. Through this split, the adapter consists of two components: The first part of the adapter, called *intermediate part*, maps from the target interface to the intermediate representation, and the second part, called *adaptee part*, maps from the intermediate representation to the adaptee interface, as shown in Figure 1. Thus, instead of providing a mapping for each concept between every language implementation, only the mapping to the intermediate representation has to be given to fully integrate it.

To create the intermediate part and to find the corresponding mapping, the target interface must be known. However, the target interface depends on the language context the adapter gets called from. During the initialization phase of the adapter, this information is not present.

To overcome this issue, the user could explicitly state the target interface if the required information is available. For example, this can be done by an explicit adaption call to the adapter before it is used.

[1] https://github.com/graalvm/graalpython
[2] org.graalvm.polyglot.proxy package (https://git.io/Jvvkz)
[3] See https://git.io/JfR0p for an example

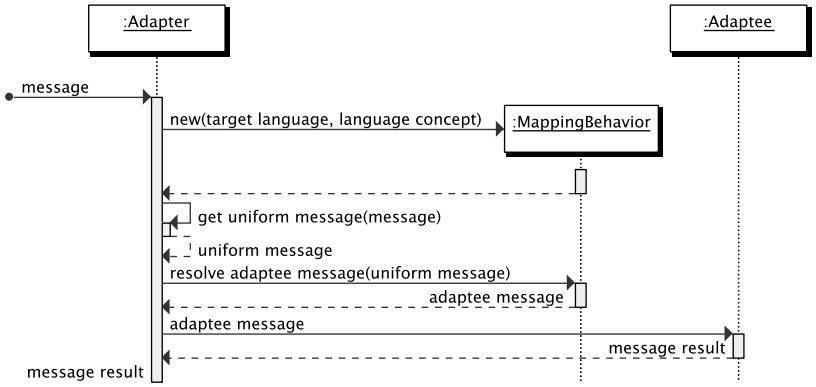

Figure 2: Sequence diagram of the adapter message resolution

While this approach provides fine-grained control, by stating everything explicitly, it can be quite cumbersome. The reason for this is primarily that, in many cases, the mapping intent is already expressed in the underlying algorithm. If, for example, a `Dictionary` is passed to a Ruby algorithm, it is clear that the target interface is at least a Ruby object and should probably also have a Ruby `Hash`-like interface. Therefore, the mapping is created during run-time, if the adapter gets called from a language context, as shown in Figure 2.

Although from a user perspective, the desired language is clear, it is not intended from the Truffle framework to be able to find out the language of an object. While, from the view of an application developer, the transparency between languages is desirable, it limits the design space for tooling developers. Therefore, as part of the implementation, such functionality was integrated into the Truffle framework.

Every time a language context is requested, a newly introduced management object saves the corresponding language. If now, a method is called on one of the adapters, the currently active language is recorded as the last caller language. Based on the language the adapter is written in, a new primitive function has to be added to access this value from a language level and use it to build the desired mapping. To identify the right interface from the given language, the language-agnostic, generic interface can specify default mappings for the different languages. Given that information, the adapter can fully automatically configure itself to the desired target interface. From an application developer's view, the boundary between the languages disappear.

4 DISCUSSION

There is always a trade-off between automatization and customization. However, making the interface translation process fully transparent to the user could result in behavior not intended by the user. Therefore, this approach tries to balance both sides.

On the one hand, the application developers are in charge of explicitly declaring an adapter with its corresponding concept. That way, the users are aware of the adapter, and overhead introduced by the adapter is prevented if not needed.

The mechanism to determine the target language, as well as the creation of the mapping behavior during run-time, introduces additional function calls and therefore performance overhead. This overhead could be further reduced by utilizing caching and reusing the mapping behaviors.

On the other hand, a lot of the process is performed automatically and, therefore, transparent for the user. Instead of using explicit conversion calls, the calls are translated automatically based on predefined configuration files. Thus, verbosity in using the adapter is reduced and makes it more accessible.

However, all non-trivial abstractions are, to some degree, leaky [11]. In the case of adapter, this appears particularly during debugging or the use of meta-programming. If, for example, the class name is requested, it is not clear whether the adapter itself or the wrapped value is meant. A similar problem occurs if the identity is requested. For these cases, a solution has to be established and consistently used across different languages.

The open nature of the configuration files enables the users to change the adapter behavior for their needs. Since every stakeholder can create, edit, and combine these configuration files, different kinds of conflicts can occur. For example, if a mapping file relies on a specific interface definition file, and that file is modified. In this case, a dependency and versioning system could help. For now, the users are responsible for keeping their configuration files consistent.

To archive this, it is conceivable to support the user in this process by providing tools. Such tools could be prototyped in GraalSqueak, the Truffle implementation of Squeak/Smalltalk, and can range from providing a specialized editor to explore and connect different interfaces, up to (fully-)automatic mapping approaches by utilizing data from the underlying source code or test suites. Furthermore, the exchange of these files can be simplified, by providing an infrastructure to upload, download, and explore these files.

Lastly, our approach is not limited to mapping interfaces of similar objects of different languages. As an example, it would be possible to create a mapping that lets a graphical object, such as a bitmap, from one language appear as a list of RGB triples in another language. Therefore, it would be interesting to explore if our approach could also be used for other purposes, such as data transformation.

5 RELATED WORK

The need to use existing data structures and objects across different runtimes has resulted in different related approaches.

Serialization. During serialization [10], an object is translated into a format that can be stored, transmitted, and later reconstructed. The advantage of this approach is that it can be integrated into nearly every language as a library. However, complex object graphs are often not supported. Since the memory is not shared between the runtimes, each object exists as a separate object in each runtime. Resulting in different identities, more memory consumption, performance overhead to transfer the object, and the need for synchronization.

In comparison, our approach preserves the identity by sharing the same object across languages and translating messages accordingly. Instead of representing a specific instance of an object in a language-agnostic format, the underlying concept itself is represented that way. Thereby, several challenges are shared between these approaches, such as schema evolution. Although this challenge is not directly addressed in this work, corresponding ideas can be borrowed from this field. Popular serialization mechanisms are, for example, Google Protocol Buffers [12], Apache Thrift [1], and Avro [4].

Polyglot Adapters. The idea to provide adapters for the GraalVM was discussed in related work [9]. It demonstrates that the adapter pattern is suitable to translate messages between languages and mimic language-specific behavior, but the approach was purely on the language level. In the course of this, scaling and usability challenges were revealed, which are tackled by this work. By introducing a language-agnostic, generic interface representation and splitting the adapter into two parts, the mapping overhead can be reduced. Further, the language resolution, which creates the adaptee part of the adapter on the fly, improves the overall usability.

6 CONCLUSION AND FUTURE WORK

In this work, user-defined interface mappings for the GraalVM are introduced. These allow developers of polyglot applications to combine algorithms and data structures from different languages more effectively by making code more portable. Thereby, a mechanism is introduced to make the whole process more transparent to the user. There is no need to adjust either the interface from the object or the calls out of the algorithm by hand. This approach enables the application developer to focus on the domain-specific challenges and using the best-fitted language concepts without being bothered with language boundaries.

This work introduces the concept of such an approach. As the next step, a concrete implementation is in planning. Building on that, the surrounding infrastructure would be an interesting starting point. Beginning from the development of tools to create and manage configuration files, up to a registry service to allow application developers to share their files in a standardized way.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of HPI's Research School[4] and the Hasso Plattner Design Thinking Research Program[5].

REFERENCES

[1] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. 2007. *Thrift: Scalable Cross-Language Services Implementation.* Technical Report. Facebook. http://thrift.apache.org/static/files/thrift-20070401.pdf

[2] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. *YAML Ain't Markup Language (YAML) (tm) Version 1.2.* Technical Report. YAML.org. http://www.yaml.org/spec/1.2/spec.html

[3] P. J. Brown (Ed.). 1977. *Software Portability: An Advanced Course.* Cambridge University Press Cambridge; New York. xiv + 328 pages.

[4] The Apache Software Foundation. 2019. *Apache Avro.* https://avro.apache.org

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, PA, USA) *(DLS 2015).* Association for Computing Machinery, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[7] Oracle Labs. 2020. *TruffleRuby – A high performance implementation of the Ruby programming language.* https://github.com/oracle/truffleruby

[8] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019).* Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[9] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. Towards Polyglot Adapters for the GraalVM. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming* (Genova, Italy) *(Programming '19).* ACM, New York, NY, USA, Article 1, 3 pages. https://doi.org/10.1145/3328433.3328458

[10] Michael Philippsen and Bernhard Haumacher. 1999. More efficient object serialization. In *Parallel and Distributed Processing*, José Rolim, Frank Mueller, Albert Y. Zomaya, Fikret Ercal, Stephan Olariu, Binoy Ravindran, Jan Gustafsson, Hiroaki Takada, Ron Olsson, Laxmikant V. Kale, Pete Beckman, Matthew Haines, Hossam ElGindy, Denis Caromel, Serge Chaumette, Geoffrey Fox, Yi Pan, Keqin Li, Tao Yang, G. Chiola, G. Conte, L. V. Mancini, Domenique Méry, Beverly Sanders, Devesh Bhatt, and Viktor Prasanna (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 718–732.

[11] Joel Spolsky. 2002. The Law of Leaky Abstractions. *Joel on Software: And on Diverse and Occasionally Related Matters* (01 2002). https://doi.org/10.1007/978-1-4302-0753-5_26

[12] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format.* Technical Report. Google. http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013).* ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[4] https://hpi.de/en/research/research-school.html
[5] https://hpi.de/en/dtrp/