Exploratory Tool-Building Platforms for Polyglot Virtual Machines

Fabio Niephaus

Exploratory Tool-Building Platforms for Polyglot Virtual Machines

Fabio Niephaus

Dissertation
zur Erlangung des Doktorgrades der
Digital Engineering Fakultät
der Universität Potsdam

Betreuer: Prof. Dr. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute
Digital Engineering Faculty
University of Potsdam
Germany

Gutachter: Prof. Dr. Elisa Gonzalez Boix
Distribution and Concurrency Research Group
Software Languages Lab
Faculty of Sciences, DINF
Vrije Universiteit Brussel
Belgium

Prof. Laurence Tratt, PhD
Software Development Team
Department of Informatics
King's College London
United Kingdom

Datum der Einreichung: 1. Dezember 2021
Datum der Disputation: 22. Juli 2022

Online veröffentlicht auf dem
Publications-Server der Universität Potsdam:
https://doi.org/10.25932/publishup-57177
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-571776

Abstract

Polyglot programming allows developers to use multiple programming languages within the same software project. While it is common to use more than one language in certain programming domains, developers also apply polyglot programming for other purposes such as to re-use software written in other languages. Although established approaches to polyglot programming come with significant limitations, for example, in terms of performance and tool support, developers still use them to be able to combine languages.

Polyglot virtual machines (VMs) such as GraalVM provide a new level of polyglot programming, allowing languages to directly interact with each other. This reduces the amount of glue code needed to combine languages, results in better performance, and enables tools such as debuggers to work across languages. However, only a little research has focused on novel tools that are designed to support developers in building software with polyglot VMs. One reason is that tool-building is often an expensive activity, another one is that polyglot VMs are still a moving target as their use cases and requirements are not yet well understood.

In this thesis, we present an approach that builds on existing self-sustaining programming systems such as Squeak/Smalltalk to enable exploratory programming, a practice for exploring and gathering software requirements, and re-use their extensive tool-building capabilities in the context of polyglot VMs. Based on TruffleSqueak, our implementation for the GraalVM, we further present five case studies that demonstrate how our approach helps tool developers to design and build tools for polyglot programming. We further show that TruffleSqueak can also be used by application developers to build and evolve polyglot applications at run-time and by language and runtime developers to understand the dynamic behavior of GraalVM languages and internals. Since our platform allows all these developers to apply polyglot programming, it can further help to better understand the advantages, use cases, requirements, and challenges of polyglot VMs. Moreover, we demonstrate that our approach can also be applied to other polyglot VMs and that insights gained through it are transferable to other programming systems.

We conclude that our research on tools for polyglot programming is an important step toward making polyglot VMs more approachable for developers in practice. With good tool support, we believe polyglot VMs can make it much more common for developers to take advantage of multiple languages and their ecosystems when building software.

Zusammenfassung

Durch Polyglottes Programmieren können Softwareentwickler:innen mehrere Programmiersprachen für das Bauen von Software verwenden. Während diese Art von Programmierung in einigen Programmierdomänen üblich ist, wenden Entwickler:innen Polyglottes Programmieren auch aus anderen Gründen an, wie zum Beispiel, um Software über Programmiersprachen hinweg wiederverwenden zu können. Obwohl die bestehenden Ansätze zum Polyglotten Programmieren mit erheblichen Einschränkungen verbunden sind, wie beispielsweise in Bezug zur Laufzeitperformance oder der Unterstützung durch Programmierwerkzeuge, werden sie dennoch von Entwickler:innen genutzt, um Sprachen kombinieren zu können.

Mehrsprachige Ausführungsumgebungen wie zum Beispiel GraalVM bieten Polyglottes Programmieren auf einer neuen Ebene an, welche es Sprachen erlaubt, direkt miteinander zu interagieren. Dadurch wird die Menge an notwendigem Glue Code beim Kombinieren von Sprachen reduziert und die Laufzeitperformance verbessert. Außerdem können Debugger und andere Programmierwerkzeuge über mehrere Sprachen hinweg verwendet werden. Jedoch hat sich bisher nur wenig wissenschaftliche Arbeit mit neuartigen Werkzeugen beschäftigt, die darauf ausgelegt sind, Entwickler:innen beim Polyglotten Programmieren mit mehrsprachigen Ausführungsumgebungen zu unterstützen. Ein Grund dafür ist, dass das Bauen von Werkzeugen üblicherweise sehr aufwendig ist. Ein anderer Grund ist, dass sich mehrsprachige Ausführungsumgebungen immer noch ständig weiterentwickeln, da ihre Anwendungsfälle und Anforderungen noch nicht ausreichend verstanden sind.

In dieser Arbeit stellen wir einen Ansatz vor, der auf selbsttragenden Programmiersystemen wie zum Beispiel Squeak/Smalltalk aufbaut, um Exploratives Programmieren, eine Praktik zum Explorieren und Erfassen von Softwareanforderungen, sowie das Wiederverwenden ihrer umfangreichen Fähigkeiten zum Bauen von Werkzeugen im Rahmen von mehrsprachigen Ausführungsumgebungen zu ermöglichen. Basierend auf TruffleSqueak, unserer Implementierung für die GraalVM, zeigen wir anhand von fünf Fallstudien, wie unser Ansatz Werkzeugentwickler:innen dabei hilft, neue Werkzeuge zum Polyglotten Programmieren zu entwerfen und zu bauen. Außerdem demonstrieren wir, dass TruffleSqueak auch von Anwendungsentwickler:innen zum Bauen und Erweitern von polyglotten Anwendungen zur Laufzeit genutzt werden kann und Sprach- sowie Laufzeitentwickler:innen dabei hilft,

das dynamische Verhalten von GraalVM-Sprachen und -Interna zu verstehen. Da unsere Plattform dabei all diesen Entwickler:innen Polyglottes Programmieren erlaubt, trägt sie außerdem dazu bei, dass Vorteile, Anwendungsfälle, Anforderungen und Herausforderungen von mehrsprachigen Ausführungsumgebungen besser verstanden werden können. Darüber hinaus zeigen wir, dass unser Ansatz auch auf andere mehrsprachige Ausführungsumgebungen angewandt werden kann und dass die Erkenntnisse, die man durch unseren Ansatz gewinnen kann, auch auf andere Programmiersysteme übertragbar sind.

Wir schlussfolgern, dass unsere Forschung an Werkzeugen zum Polyglotten Programmieren ein wichtiger Schritt ist, um mehrsprachige Ausführungsumgebungen zugänglicher für Entwickler:innen in der Praxis zu machen. Wir sind davon überzeugt, dass diese Ausführungsumgebungen mit guter Werkzeugunterstützung dazu führen können, dass Softwareentwickler:innen häufiger von den Vorteilen der Verwendung mehrerer Programmiersprachen zum Bauen von Software profitieren wollen.

Acknowledgments

I have been fortunate enough to be surrounded by many kind, generous, and smart people. Without their support, encouragement, guidance, and feedback, this work would not have been possible.

I would like to thank Robert Hirschfeld, Tim Felgentreff, and the HPI Software Architecture Group, including Tom Beckmann, Michael Haupt, Johannes Henning, Eva Krebs, Bastian Kruck, Jens Lincke, Toni Mattis, Tobias Pape, Michael Perscheid, Patrick Rein, Stefan Ramson, Matthias Springer, Marcel Taeumel, Marcel Weiher, and the many students I have worked with.

I am very grateful for the support from the HPI Research School and the Hasso Plattner Institute, including Prof. Dr. Tobias Friedrich, Prof. Dr. Christoph Meinel, Prof. Dr. Andreas Polze, and Sabine Wagner, as well as from Mario Wolczko and Oracle Labs, including Christian Humer, Thomas Würthinger, Eric Sedlar, and many others from the GraalVM project.

I would further like to thank Elisa Gonzalez Boix, Laurence Tratt, Michael Van De Vanter, and many others from the programming language and virtual machine research communities, including Lars Bak, Edd Barrett, Carl Friedrich Bolz-Tereick, Alan Borning, Gilad Bracha, Matthew Flatt, Richard P. Gabriel, Antony Hosking, Stefan Marr, Hidehiko Masuhara, and Chris Seaton.

Moreover, I want to thank the Smalltalk community, including Clément Bera, Vanessa Freudenberg, Dan Ingalls, Alan Kay, Craig Latta, Florin Mateoc, Eliot Miranda, Yoshiki Ohshima, Hernan Wilkinson, and many others.

Finally, I would like to thank my best friend and partner Nicole Loyeck, my family, and my friends, who always believe in me and support me in everything I do.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AOT	ahead-of-time
API	application programming interface
AST	abstract syntax tree
AWFY	Are We Fast Yet
CFFI	C-based foreign function interface
CLR	Common Language Runtime
DAP	Debug Adapter Protocol
DSL	domain-specific language
FFI	foreign function interface
GC	garbage collector
IDE	integrated development environment
IPC	inter-process communication
IR	intermediate representation
JIT	just-in-time
JSON	JavaScript Object Notation
JVM	Java HotSpot Virtual Machine
LSP	Language Server Protocol
OOP	object-oriented programming
OS	operating system
REPL	read-eval-print-loop
SLOC	source lines of code
SSPS	self-sustaining programming system
UI	user interface
VM	virtual machine

Part I.

Programming in a World of Many Languages

1. Introduction

Today, there are many programming languages for many different purposes that developers can use for building software. Each language usually not only comes with its own syntax and semantics but also with its own set of libraries and frameworks as well as different programming tools. Learning all of this takes considerable time.

Even though general-purpose programming languages, such as C++, Java, or Python, are designed to build a wide variety of different types of applications, there are none that fit best in all situations. Software written in one language can often only make use of a single language ecosystem. Developers, however, sometimes would like to re-use and combine the knowledge and experience they had to acquire for each language they have previously worked with.

Polyglot programming is the practice of writing software in multiple programming languages within the same software project. In some programming domains, it is common to work with different languages at the same time. Prime examples are database applications. They typically make use of a data language such as SQL for accessing a database, while the rest of the application is written in another, often general-purpose language such as Java. In other domains, polyglot programming may not be common practice. Nonetheless, the sheer number of libraries and frameworks written in different languages and for many different domains and purposes often make it hard to decide on a particular language. Therefore, developers would often like to be able to use more than one language in their software applications.

Different language integration techniques exist that allow developers to combine multiple languages: Foreign function interfaces (FFIs), for example, allow developers to call out to routines written in other programming languages. They are often used to connect low-level application programming interfaces (APIs) or to accelerate high-level languages with more efficient, low-level code. Similarly, inter-process communication (IPC) allows subcomponents of a software system running in different processes to communicate with each other independently from the languages they are written in.

The different techniques for language integration, however, also come with different limitations: They often use a lower-level abstraction, the operating system (OS), or a network connection for example. These abstractions require non-trivial glue code and can cause performance overheads, data duplication, and data synchronization problems. Furthermore, programming

tools are often unable to see through low-level abstractions because they usually operate on the language level. As a consequence, developers have to switch between tools whenever they switch between languages, which adds additional cognitive overhead and increases the potential for errors. In some cases, developers can only use OS-level tools that have no understanding of higher-level languages to analyze language integration problems.

Polyglot virtual machines (VMs) provide a new approach to polyglot programming. These virtual machines allow the execution of code written in different programming languages, often referred to as *guest* languages, within the same VM instance. By avoiding low-level abstractions, they can provide high-level language interoperability, which in turn allows direct exchange of data, objects, and messages between their guest languages. As a result, less glue code is required for the integration and more code from other languages can be re-used. At the same time, developers can be better supported through tools that work across all guest languages. Overall, polyglot VMs have a clear goal: increase developer productivity by allowing developers to re-use and write code across multiple languages.

While programming with polyglot VMs has many advantages for developers and can potentially solve some development problems in new ways, it also provides new opportunities and faces new challenges:

Many mainstream tools for software development already come with support for multiple languages. General-purpose code editors, for example, allow developers to write code in different languages. However, the majority of tools with multi-language support assume that only one language is used at a time. To help developers to build polyglot applications, existing tools must be extended to support multiple languages at the same time. Since polyglot VMs allow tools to operate across languages, there is an opportunity to further support polyglot programming with new tools that are specifically designed for it. Extensions for existing tools and new tooling ideas must be explored somehow.

Since language interoperability as provided by polyglot VMs is dynamic and can thus best be observed at run-time, tools based on static analysis [117] are put at a disadvantage. Moreover, these static tools are usually limited to a fixed number of languages, requiring additional work to support each language. Polyglot VMs can provide tools based on dynamic run-time data that are language-agnostic, which allows them to operate across all existing and future guest languages of a polyglot VM [210]. These dynamic tools must, however, usually communicate through dedicated tool interfaces provided by polyglot VMs, which are commonly designed for debugging and monitoring purposes and, therefore, limit the scope of exploration.

Dynamic tools built in a language-agnostic way are a good step forward, especially for polyglot programming which provides language interoperability in a dynamic way. However, these tools can also only provide language-agnostic views. To understand dynamic behavior and state in polyglot applications, tools must be extended so that they are aware of a polyglot VM. This way, they can, for example, provide additional information on language interoperability to help developers to understand the interaction of languages or to reason about multiple language semantics at the same time.

Apart from this, polyglot programming with polyglot VMs is relatively new and not yet as well-understood as C-based foreign function interfaces (CFFIs), IPC, and other approaches to polyglot programming. More data points are needed to better understand and showcase how developers can benefit from polyglot VMs. Example applications help to demonstrate how programming with polyglot VMs can be applied and may convince developers that polyglot VMs are viable foundations that allow them to build polyglot production systems.

Self-sustaining programming systems (SSPSs) such as Smalltalk [52], Lively Kernel [74], or Self [208] provide two means that are well-suited to explore tooling ideas for polyglot programming: On the one hand, they come with tools for exploratory programming [172, 175, 203]. These tools are designed to help developers explore and gather requirements interactively for the software they want to build. With them, we could, for example, interact with different languages at run-time or explore sources for dynamic run-time data that could be useful in tools in detail. On the other hand, they support rapid tool-building through live programming [156]. Just like other applications and parts of the system, tools can be built and evolved at run-time with short feedback loops. However, self-sustaining programming systems and their exploratory tools are usually based on a particular programming language and designed for exploration within the system they belong to, not necessarily for exploration of external components such as other languages or a polyglot virtual machine.

In this thesis, we present an approach that allows the reuse of tools for exploratory programming across the guest languages of a polyglot VM and the VM itself. We show that the tools of an existing self-sustaining programming system can be re-used with little to no modification. Since they only access information through reflection, the reflection interface of an SSPS's object protocol must be redirected through the language interoperability protocol of a polyglot VM for them to operate across languages. The implementation effort for this is low and more importantly, these redirections must be implemented only once for the exploratory tools to work for all existing and future languages of a polyglot VM.

While such tools may already be useful for the exploration of language interoperability and polyglot VMs, they may not always provide enough context about the languages used in polyglot applications as they were originally designed for one particular language. For this reason, we further propose extensions for these tools that make them aware of the polyglot VM they run on. This way, they provide additional information in the presence of multiple languages to better support polyglot programming.

Moreover, we demonstrate that polyglot programming can also be applied to tool-building and within an SSPS. This not only makes tool-building more productive because more software can be re-used and can help to improve SSPSs. It also provides more insights into programming with polyglot VMs and helps us to better understand advantages and challenges.

With TruffleSqueak, we present an implementation of our approach based on Squeak/Smalltalk, an open-source self-sustaining programming system. It provides tools for exploratory programming, short feedback loops through live programming, and extensive tool-building capabilities for and hosted on top of the GraalVM, a state-of-the-art polyglot virtual machine. TruffleSqueak's exploratory tools allow users to explore high-level interactions between GraalVM languages, all the way down to the level of its protocol for language interoperability, the host language, and the runtime system. Moreover, we can use the tool-building infrastructure of Squeak/Smalltalk to prototype and build new tools and applications in a polyglot way.

Further, we evaluate both our approach and our implementation. We assess the compatibility and the run-time performance of TruffleSqueak and show that it fulfills the requirements of our approach. With the aid of several case studies, we illustrate how different ideas for polyglot applications and new tools for polyglot programming can be explored with our platform. Moreover, we demonstrate that our approach can also be applied to other polyglot VMs and that insights gained through a platform like TruffleSqueak can further be transferred to other programming systems. We discuss our observations and insights, compare our research with related work, provide an outlook of future work, and end the thesis with a conclusion.

[|]

Thesis Statement To use and evolve polyglot virtual machines effectively, we must be able to explore tooling ideas, polyglot applications, language implementations, and the VMs themselves at run-time.

1.1. Challenges

While polyglot virtual machines provide a new level of polyglot programming that comes with many advantages over established approaches, they also come with new challenges. In the following, we highlight some of them and explain how they are addressed in this thesis.

Tools for Polyglot Programming With Polyglot VMs

Developers need and use tools to build software systems. They can choose from a wide variety of tools that support different tasks during the software development process. Most of the tools that developers use, however, are designed to support one programming language at a time. This is also true for general-purpose tools that support multiple languages individually, but not their integration to support polyglot programming.

Polyglot programming with CFFIs and IPC use OS-provided abstractions that are usually boundaries for tools. As a consequence, developers often have to switch between different sets of tools when using multiple languages through such language integration techniques. Polyglot VMs avoid these abstractions and allow tools to operate across their guest languages at the same time. These VMs are, however, relatively new compared with CFFIs and IPC and, therefore, there are not many tools that help developers to build polyglot applications that make use of them.

Figure 1.1 illustrates how many developers build software in general and how this looks like when building polyglot applications. They often distinguish between *development-time*, during which they write the software, and *run-time*, during which their applications run. Development usually happens in an edit-compile-run cycle. At development-time, developers often use *static* tools such as code editors, for example, to read and write code. These tools perform static code analysis [117] to gather the information they need for syntax highlighting, code completion, and other features they provide. *Dynamic* tools such as debuggers, on the other hand, are connected with the VM and have access to dynamic run-time data containing information on concrete program behavior and state. This data can only be observed and captured during the execution of an application. However, many developers often only use dynamic tools to debug and monitor running applications, not to build them.

Although there has been research on static tools for polyglot programming (e.g., [35, 97, 103]), these tools often only support a fixed number of languages. To add another language, their static analysis must be extended for that language as well as for its integration with all other languages (① in Figure 1.1).

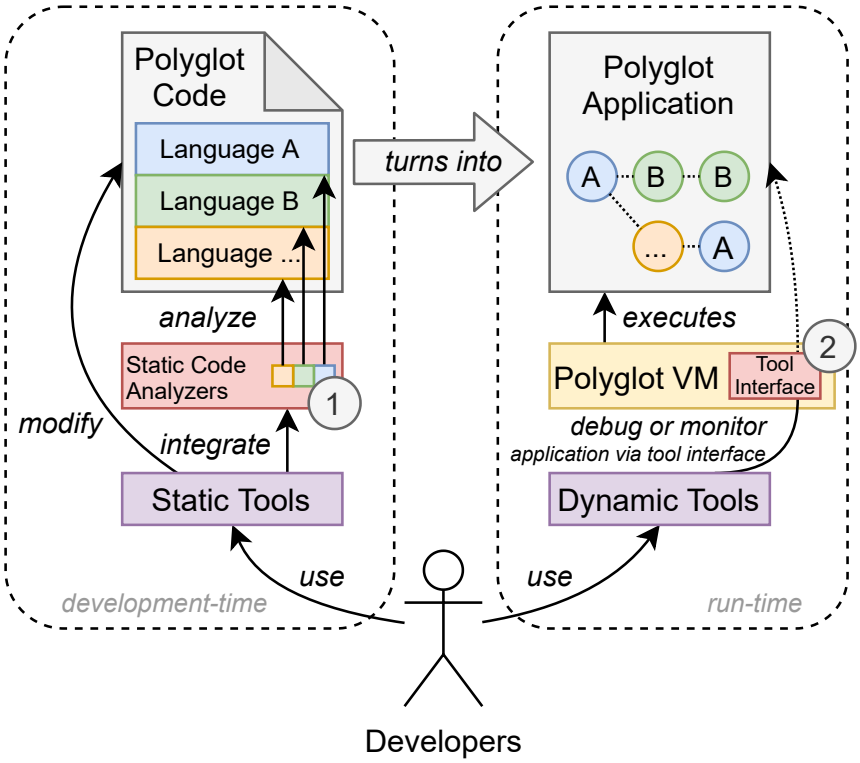


Figure 1.1.: Developers often distinguish between development-time and run-time. This is often reflected by the types of tools they commonly use: static tools for development and dynamic tools at run-time. Static tools such as code editors usually base their assistive features on static code analysis. For polyglot code, they typically support multiple languages and their combinations (①). Dynamic tools such as debuggers and profilers allow developers to inspect their applications at run-time. For this, they usually interact with the VM through an appropriate interface (②). Such dynamic tools are, however, restricted to the capabilities of these tool interfaces, which are usually designed for debugging and monitoring purposes. These tools can thus not directly interact with running applications and other components of a polyglot VM.

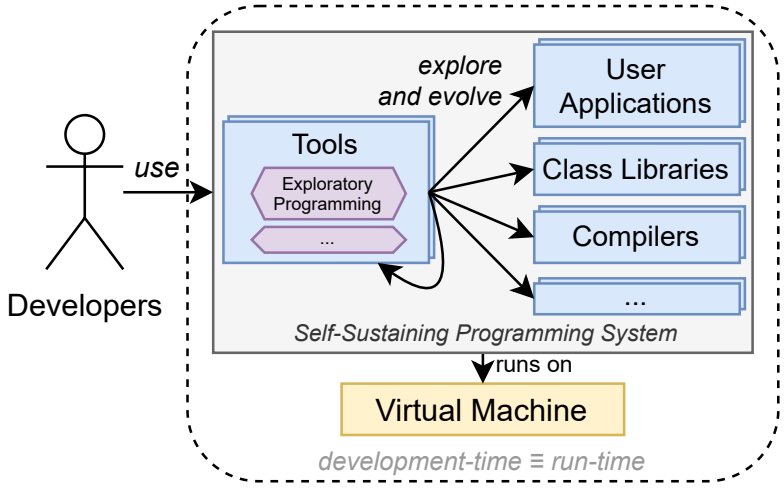


Figure 1.2.: Self-sustaining programming systems do not distinguish between development-time and run-time. Their tools run on the same VM alongside all other components of the SSPS. As a result, different tools, such as tools for exploratory programming, can explore and modify not only user applications but also themselves, other tools, class libraries, compilers, and all other components at run-time. This way, the SSPS can be evolved over time.

How languages are integrated, however, depends on the polyglot VM that is used to execute a polyglot application.

In this thesis, we focus on dynamic tools, which can be built based on infrastructures provided by polyglot VMs, which allow them to work across all existing and future guest languages [210]. Similar to other VMs, polyglot VMs provide dedicated tool interfaces for debuggers, profilers, and other dynamic tools (② in Figure 1.1). These tool interfaces, however, clearly separate tools from applications and are often geared towards debugging and monitoring purposes, which limits the exploration space for tooling ideas. Monitoring tools such as samplers typically only observe running applications. And during debugging, developers can only interact with and change applications when they are paused.

Although for different reasons, ideas for both static and dynamic tools that, for example, support the construction of polyglot applications can thus be difficult to explore in the context of polyglot VMs. Nonetheless, developers can benefit from both kinds of tools for polyglot programming throughout the development process.

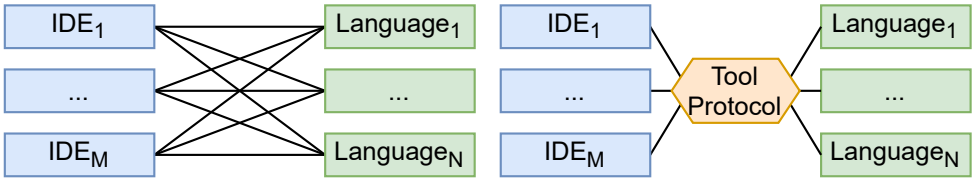
In self-sustaining programming systems such as Smalltalk, Lively Kernel, or Self, development always happens at run-time as illustrated in Figure 1.2. Therefore, there is no distinction between development-time and run-time. Their support for live programming shortens the development cycle as appli-

cations can be evolved while they are running [156]. This makes tool-building more productive because tools are also just applications from the perspective of the system. Moreover, these systems are built in such a way that they can even evolve themselves. Most of their components are written in the language they use, starting with the language's compiler and class libraries all the way up to user applications. They also provide many reflective facilities that are not limited to introspection [44]. Through intercession [86], it is possible to manipulate any object and meta-level data structure such as classes, methods, and even stack frames. Dynamic run-time data is, therefore, always accessible and editable, and can be used in tools.

Moreover, self-sustaining programming systems usually provide tools for *exploratory programming*. These tools allow evaluation of code as well as inspection and modification of objects at run-time in an interactive way. Exploratory tools are designed to help developers explore and gather requirements for the software they want to build and to better understand the dynamic behavior of their applications. This makes them a good fit for exploring tooling ideas for polyglot programming, allowing us, for example, to interact with different languages at run-time or to explore different sources for dynamic run-time data in detail.

In this thesis, we present an approach that enables exploratory programming and tool-building on top of polyglot VMs. Instead of creating a platform for exploratory programming and tool-building from scratch, we propose to build on an existing SSPS. We show what is required to adapt the exploratory tools of such a system so that they can be re-used across all guest languages of a polyglot VM. The effort to adapt these tools is low and needs to be done only once. We illustrate what APIs polyglot VMs need to provide to implement this. As a result, adapted exploratory tools can be used for any language of a polyglot VM, including its host language and future guest languages. At the same time, they can directly interact with applications at run-time. Exploration is thus no longer restricted to the capabilities of the tool interface provided by a polyglot VM.

With a platform like this, we can explore new ideas for tools that help developers to build polyglot applications. We further show that the platform can also help other developers. Since language implementations and other components of a polyglot VM are not implemented as part of the SSPS, it may not be possible to evolve them at run-time. Nonetheless, our platform can also help language and runtime developers to explore the components of polyglot VMs while they are running, which can provide insights into how they can be evolved in general.



(a) State of the art: To support N programming languages in M IDEs, $M * N$ adapters are required.

(b) Tool protocols reduce the complexity to support N languages in M IDEs to $M + N$.

Figure 1.3.: The complexity of supporting N languages in M IDEs without and with a common protocol for tools.

The Need for Polyglot-Aware Tools

Many tools for software development are designed for one specific language, often because they are built by the maintainers or the community of that language. As such, they can provide common features but also language-specific ones. Developers, however, must learn how to use these tools for each language. Tools often have different user interfaces (UIs) and can provide similar features in different ways. To reduce this burden for developers, general-purpose integrated development environments (IDEs) come with tools that support multiple programming languages. This way, developers can use one familiar set of tools for developing code in different languages.

However, the construction of such general-purpose IDEs faces scalability issues. As illustrated in Figure 1.3a, $M * N$ adapters are needed to support N languages in M IDEs. Keidel, Pfeiffer, and Erdweg call this the *IDE portability problem* [83]. Since each adapter is specific to a particular language, there is only little potential for code reuse.

In recent years, there have been several efforts to decouple IDEs and tools from languages with the goal to reduce this complexity. Figure 1.3b illustrates how a common protocol for tools can reduce the complexity to support multiple languages in IDEs to $M + N$. The Language Server Protocol (LSP) [111], for example, is such a protocol and supports code navigation, code completion, and other IDE features. For each language, a language-specific server provides these features through the protocol. Developers can thus use their preferred IDE to develop in different languages, given that the IDE has support for the LSP and that there are appropriate language servers. The Debug Adapter Protocol (DAP) [110] works in a similar way but focuses on debugging. It decouples debugging UIs from specific debugging interfaces provided by different language runtimes. Monto [83] is another, more generic protocol that decouples IDEs from languages.

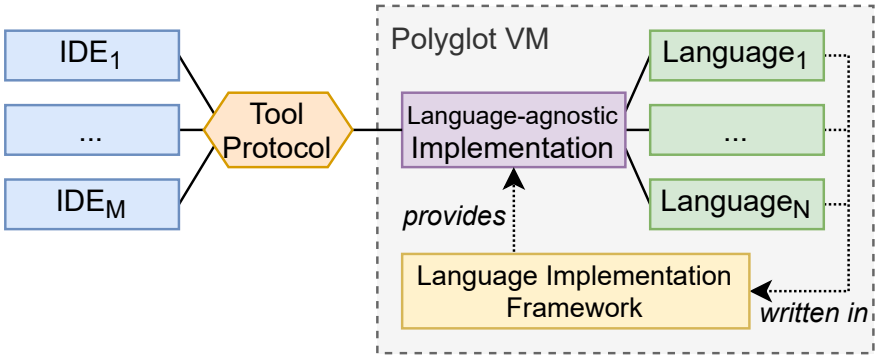


Figure 1.4.: In polyglot VMs, dynamic tools and even entire tool protocols can be implemented on top of language implementation frameworks. These language-agnostic tools reduce implementation efforts and can operate across guest languages at the same time. Without appropriate extensions, however, they usually only provide language-agnostic views.

Although general-purpose IDEs support multiple languages, they usually assume that only one is used at a time, independently of whether they deploy language-specific adapters or use a protocol for tools. This assumption, however, no longer holds for polyglot programming, which allows multiple languages to be used simultaneously. Polyglot VMs often use a common intermediate representation (IR) across the languages they support. Such IRs can be used to provide language interoperability but also to build dynamic tools that work across languages.

Van De Vanter *et al.* [210] have presented an approach that takes this idea even further. With an appropriate tooling infrastructure in the language implementation framework of a polyglot VM, it is possible to build dynamic tools independently from specific languages as shown in Figure 1.4. Instead of implementing tools or entire tool protocols for each language individually, language-agnostic implementations on top of the language implementation framework can be shared across guest languages, which reduces implementation efforts. Therefore, language developers only need to provide language-specific functionality required by the tooling infrastructure to unlock several tools for their language. This approach has been applied in GraalVM to build various dynamic tools. As an example, GraalVM provides an implementation of the DAP written in its Truffle framework. GraalVM languages only need to provide language-specific debugging functionality. The implementation of the server and other sharable components are provided by Truffle.

While such language-agnostic tools are a good step forward, they can also only provide language-agnostic views. In debuggers and other tools that incorporate source code or files, this problem may not be obvious. GraalVM's

support for the DAP, for example, allows developers to step through code written in different languages. Since debuggers display source code for stack frames, developers usually know what language is being executed. When inspecting variables and values stored in scopes, however, developers are presented with a language-agnostic view that is not linked to source code. Therefore, the view does not help developers to understand from what languages specific objects come from. This makes it hard to understand polyglot data structures and other complex objects in polyglot applications as well as the interaction between them.

In general, tools that were initially designed to support one language fail to support developers whenever dynamic program behavior and run-time state within polyglot applications need to be understood in detail. In these situations, it is often hard to map dynamic behavior and state to specific source code locations. To better support developers to use polyglot programming, we thus believe that tools need to be extended so that they are aware of the polyglot VM. We call such tools *polyglot-aware* tools.

As part of this work, we show how to enable exploratory programming on top of polyglot VMs. One of the goals of exploratory programming is that the means of exploration must always be explainable. In a first step, we show that existing tools for exploratory programming can be re-used for other languages. Since these tools were also initially built for one language, we propose extensions that make them polyglot-aware. With these extensions, they can, for example, always provide sufficient information on the language, structure, interfaces, and specific interoperability features of any object.

Our approach further enables the construction of new tools for polyglot programming. Since we build on an existing self-sustaining programming system, we can use its extensive tool-building capabilities to explore different ways of providing context for polyglot programming in tools, including its tools for exploratory programming and other existing tools. As we demonstrate in different case studies, the infrastructure required for our extensions can also be used to build entirely new polyglot-aware tools.

Making Effective Use of Programming With Polyglot VMs

Although programming with polyglot virtual machines has many advantages over established approaches to polyglot programming, it has not yet been widely applied in production systems. For one, polyglot VMs are relatively new and still rapidly evolving because their requirements, such as the ones for language interoperability, have not yet been fully understood. Further exploration of different use cases and ideas is needed.

Moreover, only a few polyglot VMs provide a high level of compatibility across all languages they support. Language compatibility, however, is an important factor for developers. If alternative language implementations of polyglot VMs are not fully compatible with the corresponding reference runtime, some libraries, frameworks, or language features may not be available for development. Finding out which parts work and which do not is additional work that developers would like to avoid.

Many alternative language implementations struggle to provide and retain full compatibility with low-level C extensions. These extensions are often used when high-level languages fail to provide good run-time performance for performance-critical operations. In the case of Python, for example, CPython, its reference VM, is known to be slow compared with VMs for other languages (e.g., [4, 101]). Therefore, Python libraries and frameworks, such as Tensorflow and NumPy, heavily rely on C code to provide competitive performance. PyPy, GraalPython, and other alternative VMs for Python can significantly improve the run-time performance of Python code through just-in-time (JIT) compilation. JIT compilers, however, perform best if most of the code they should optimize is visible to them and not hidden in a shared library. Calling out to shared libraries written in low-level C code is, therefore, often counterproductive but it is something that established language integration techniques have led to. This is an example where there is a clear difference between programming with polyglot VMs and other techniques for language integration. Differences like this must be both well-understood and well-communicated to developers to allow them to use polyglot VMs effectively.

Apart from that, full language compatibility is rarely needed to support the majority of applications, libraries, and frameworks written in a particular language. Another, maybe more important reason for the slow adoption of polyglot VMs is that established language integration techniques have influenced the way developers think about polyglot programming. Since language integrations usually introduce performance overheads, developers may think that using multiple languages slows down their applications. Or they may tend to structure their code in such a way that the number of cross-language calls is kept to a minimum to avoid these performance overheads. Furthermore, callbacks are often poorly supported by language integration techniques. Consequently, developers may think that libraries are easier to re-use than frameworks.

Similarly, developers may refrain from mixing languages because they think that language integrations are poorly supported by their tools. For example, they may decide against using low-level C code in their Python application, because they do not want to give up the programming experience they are used

to from Python tools such as the debugger. Developers are often not aware that polyglot VMs can provide better tool support and thus a better programming experience compared with other language integration techniques. Ultimately, their goal is to increase software reuse and to allow developers to always use the "best" language, framework, library, or tool for the task without sacrificing performance or tool support. However, just because something is possible and better, in theory, does not mean that developers will use this new level of polyglot programming in practice, let alone that companies are willing to run such polyglot applications in production. More practical experiences on polyglot programming with polyglot VMs are needed to streamline APIs and tools for developers, distill best practices, and make this type of programming more approachable in practice.

The approach presented in this thesis builds on an existing self-sustaining programming system and makes its tools for exploratory programming available to the languages of a polyglot VM. This makes it possible to use them not only to explore polyglot user applications but also tools, language implementations, and the internals of the polyglot VM they run on. Moreover, SSPSs allow developers to extend existing tools for specific needs such as new capabilities for polyglot programming. Since these systems typically provide extensive tool-building capabilities, they can also be used to build entirely new tools that are designed for building polyglot applications for example.

We show that polyglot programming can also be applied to tool-building to increase software reuse and to improve the productivity of tool developers. This way, for example, new tools for polyglot programming can themselves be built in a polyglot way. Polyglot programming can even be expanded to an entire self-sustaining programming system. Every component written in the language of the system is a user application from the perspective of the polyglot VM and can thus make use of all supported languages. While this can improve reuse and productivity, such efforts also help to gather new data points and practical experience on polyglot programming.

We present case studies of several polyglot tools and applications built in TruffleSqueak, an implementation of our approach for the GraalVM. In one of these studies, we also explore how TruffleSqueak itself can be extended with a polyglot drawing backend. We report lessons learned and insights on polyglot programming with polyglot VMs for each study based on our practical experiences. With another case study, we further show that these learnings and insights are not limited to our platform and can also be transferred to other programming systems. While some of our case studies explore specific advantages or challenges of programming with polyglot VMs, we believe others demonstrate the potential of polyglot VMs and possibly inspire other developers to create more polyglot applications.

1.2. Contributions

The main contributions of this thesis are as follows:

1. An approach enabling exploratory programming and tool-building on top of polyglot virtual machines.
2. Extensions for exploratory programming tools that make them polyglot-aware.
3. A proposal to further explore polyglot programming by applying it to tool-building and within a self-sustaining programming system.
4. An implementation of said approach and extensions for the GraalVM and based on Squeak/Smalltalk.
5. Case studies that demonstrate how our approach enables further research on tools for polyglot programming and polyglot virtual machines, as well as a synthesis of our findings.

1.3. Outline

The remainder of this thesis is structured as follows: In Chapter 2, we illustrate the state-of-the-art of polyglot programming. Then, in Part II, we introduce programming languages, virtual machines, developer tools, and GraalVM, a state-of-the-art polyglot VM. We present our approach in Part III and show an implementation of it in Part IV. We evaluate our approach and implementation in Part V. Finally, in Part VI, we discuss our observations and insights, compare our research with related work, provide an outlook of future work, and conclude the thesis.

2. State of the Art of Polyglot Programming

This chapter gives an overview of the state of the art of polyglot programming. We discuss different motivations for polyglot programming, describe established approaches that enable it, and elaborate on the opportunities of polyglot VMs. Our work builds on previous work on polyglot VMs and aims at advancing the level of polyglot programming they provide.

2.1. Use Cases

Today, developers can build software with thousands of different programming languages. Languages provide different programming paradigms and levels of abstraction, have different performance characteristics, and suit some programming domains better than others (e.g., [108, 149, 151, 171, 174]).

Low-level programming languages such as C, for example, provide few abstractions and are thus closer to the hardware. While having fewer abstractions often translates to more work for developers, developers can use low-level languages to write highly efficient code for specific hardware platforms.

High-level programming languages, on the other hand, provide rich abstractions that allow developers to write, for example, more expressive, secure, or portable code and, therefore, can increase productivity. Java, for example, provides automatic garbage collection, memory safety, and a type system among others and since it is interpreted by a VM, code written in Java is portable across different hardware platforms.

Domain-specific languages (DSLs) go even further in terms of abstractions: They are specifically designed for particular domains and allow developers to write very expressive code [107]. SQL, for example, is designed for writing database queries. However, writing complex algorithms in SQL is often cumbersome, sometimes even impossible.

Moreover, some languages are particularly common in certain programming domains. The Python language, for example, is very popular for scientific computing [133]. Different programming domains frequently evaluate how suitable different languages are for their needs (e.g., [3, 14, 32, 45, 76, 176]).

In addition, new trends influence what kind of languages developers prefer for certain programming domains [26, 147]. JavaScript, for example, started out as and still is the programming language used in web browsers. With Node.js, however, JavaScript has also become very popular for writing server applications.

As a result, developers have different reasons to use polyglot programming: Programming domains can overlap, especially when building complex software systems. Polyglot programming allows developers to write different parts of their applications in the languages they prefer. Popular examples for this are database applications. These applications often use SQL for querying the database, while the business logic and other parts of the application are written in, for example, a general-purpose language such as Java.

A related motivation is software reuse [90]. Using a single language means that developers can only build on the libraries and frameworks available in that language. If they want to re-use software from another language, they usually have two choices: a) port the software to their language, which can be a time-consuming and, therefore, expensive activity, or b) integrate the existing software into their codebase through polyglot programming. Moreover, and with every additional language they can use within their project, the entire repertoire of libraries and frameworks written in that language becomes accessible and thus re-usable. Developers building, for example, server applications with Node.js in JavaScript, may for example use libraries such as NumPy created in Python by the scientific computing community for number crunching.

Another reason for using multiple languages within software projects is related to different performance characteristics of languages: While programming with high-level programming languages such as Python or Ruby can be more productive, their abstractions can sometimes introduce significant performance overheads. Therefore, developers using high-level languages sometimes want to accelerate performance-critical parts of their applications with more efficient, low-level code. For this reason, VMs for Python and Ruby, for example, provide dedicated means for integrating C or C++ code [10, 153, 170].

Apart from that, there are many other reasons for using polyglot programming. Sometimes, the software development process leads to it, for example, when different developer teams work with different languages. In other cases, parts of a software project can evolve over time into a DSL or some other form of programming language. External factors, such as customer requirements, can also lead to the use of polyglot programming. Some computer games, for example, can be extended by end-users through appropriate programming interfaces using different languages. Other examples are REST APIs

or pre-compiled libraries that developers can use to interact with an external service independently from the language the service is written in. Similarly, software vendors sometimes provide software development kits in different languages to interact with their services. Therefore, developers sometimes even use polyglot programming unknowingly in some form.

2.2. Established Language Integration Techniques

As the previous section has illustrated, developers have an increasing need to combine languages. Interoperability between languages has therefore become more and more important [27]. This section gives an overview of commonly used techniques for language integration, provides examples of how they are used, and shows up their limitations. This helps to better understand the potential of polyglot VMs as well as the contributions of this work.

C-Based Foreign Function Interfaces FFIs generally allow programming languages to call out to functions written in other languages. While the term FFI can generally be used to describe any kind of language interoperability interface, most implementations present in many VMs for high-level programming languages commonly target native code written, for example, in C or C++. Examples are Python's CFFI [164], the `Fiddle` module in Ruby [169], or the Java Native Interface [135]. These CFFIs are often used to interact with low-level APIs from the operating system, to connect database drivers and other components typically written in low-level languages, to accelerate interpreted languages with C code, and to re-use functionalities provided by shared libraries.

To execute foreign functions, CFFIs leverage a mechanism provided by the operating system to load shared libraries into the VM process executing the high-level language. This can be done using the `dlopen` system call on Unix systems and `LoadLibrary` on Windows. Once a library is loaded, it is possible to lookup pointers for functions and other data. To call such a function, the VM needs to marshal high-level language arguments so that they match the calling convention of the underlying architecture.

To connect shared libraries with a codebase, non-trivial glue code is usually required. This glue code must often be written by the developer, which can distract from the actual work and imposes a maintenance burden. Some of this work, however, can also be automated with SWIG [9] and other tools [159] that help to generate such code. For some libraries, developers can use existing language bindings that provide the necessary glue code to connect these libraries with particular languages. For Tensorflow, a popular machine

learning framework, there are several language bindings, for example, for Java [199].

Furthermore, some language VMs provide shared libraries to interact with them on the C level. By embedding the shared library from a VM for language B into another for language A, it is possible to allow language A to call into low-level code but also to execute code written in language B. PyCall.rb [115], for example, embeds CPython into Ruby, PyCall.jl [79] embeds it into the Julia language. ExecJS [191] can embed different JavaScript VMs into Ruby and rpy2 [168] is a binding for R in Python.

However, CFFIs also come with significant limitations: Apart from the need for glue code, they usually impose performance overheads [190]. CFFIs can also lead to memory leaks and other interferences with garbage collectors (GCs) [31]. More importantly for developers, however, is that CFFI calls are hard to debug. Debuggers of high-level languages usually cannot step into native code, which means that developers often also need to use low-level, OS-provided tools such as `gdb`. Moreover, errors in use can cause invalid memory accesses, which in turn often result in crashes and give developers no chance to recover these errors at run-time.

Languages Built on Top of Others Another common technique to integrate languages is to build them on top of others. This can be done, for example, by implementing an interpreter for a language on top of another. Such interpreters often provide interoperability between the guest language and their host language. JRuby [130], for example, is a Ruby implementation written in Java and was built specifically to allow interoperability between Ruby and Java based on the Java HotSpot Virtual Machine (JVM). MagLev [184], on the other hand, is a Ruby written in Smalltalk and runs on the GemStone/S VM.

Instead of an interpreter, languages can also run on top of others based on source-to-source translation. Whalesong [224], for example, is a compiler that translates Racket to JavaScript and provides interoperability between them. Similarly, a Smalltalk implementation in Self is based on source-to-source translation and allows communication between the two languages [217] Amber [2], on the other hand, is a Smalltalk dialect and translates Smalltalk code to JavaScript.

Moreover, the translation of source code can also target other IRs such as abstract syntax trees (ASTs) or bytecode [28]. Racket, for example, provides an API for building languages on top of it based on AST transformation [201]. JVM-based languages, such as Scala [132] or Groovy [87], compile to Java bytecode and provide interoperability with Java. Java bytecode is generally a popular compilation target also used by compilers for many other languages such as, for example, Scheme [16] or Standard ML [11]. WebAssembly [59]

and LLVM bitcode [93] are two instruction formats designed to accommodate a wide variety of programming languages and can, therefore, also be used as shared IRs to combine languages and to facilitate interoperability between them.

Languages built on top of others typically avoid low-level abstractions. As a result, languages can interoperate on a higher level, which usually reduces the amount of glue code required compared with CFFIs. This also means that debuggers and other tools, at least those of the host language, can be used across languages. To improve tools for guest languages, additional work is often required. Source maps, for example, allow debuggers to map the execution of translated code back to the origin language. In addition, the runtime performance of a guest language usually depends on the host language.

This type of language integration is, however, often limited to two languages: the guest and the host language. Therefore, developers can usually only make use of libraries and frameworks from two specific language ecosystems.

Inter-Process Communication Operating systems provide several mechanisms for IPC, such as files, sockets, and shared memory, that allow different processes to communicate with each other [57]. While these mechanisms are often used to build distributed systems, they can also enable interoperability between languages [213]. A simple example of this is the PythonBridge for Pharo, which allows the execution of Python code from Smalltalk using a Python-based server through a network connection [19]. With interface description languages [91], as used in CORBA [131], Apache Thrift [178], or Google's Protocol Buffers [54] among others, more complex systems can be built with different languages and on top of IPC mechanisms. These description languages allow developers to define data structures and interfaces independently from specific languages. Based on such definitions, concrete definitions for specific languages and appropriate serialization code can be generated and then used on top of an IPC mechanism.

This approach, however, has several constraints: First of all, it enforces a distributed architecture, which may not be suitable or overcomplicated for some polyglot applications. In addition, remote procedure calls and other means for IPC impose performance overheads. Objects and messages need to be serialized, transmitted, deserialized, and are often duplicated across multiple processes. Data duplication not only increases the overall consumption of memory and disk resources. It also requires synchronization in case two or more processes operate on the same piece of shared data at the same time. Due to the distributed architecture, debugging across multiple processes that are part of an IPC-based polyglot application can be cumbersome: Similar to

FFI calls, high-level language debuggers are usually unable to step through a remote procedure call or some other mechanism and into the code written in some other language running in some other process.

2.3. Polyglot Virtual Machines

Polyglot VMs such as .NET's Common Language Runtime (CLR) [18] and GraalVM [222] are virtual machines designed to support multiple programming languages and typically provide interoperability between the languages they support. Although the CLR and GraalVM are among the most advanced polyglot VMs, there are several other polyglot VMs such as the Portable Common Runtime [214], the Virtual Virtual Machine [43], Seam [20], Mote Runner [22], Smalltalk/X [66], or Unipycation [6].

In polyglot VMs, common VM components, such as garbage collector and JIT compilers among others, can be re-used across multiple languages. This reduces the burden of maintaining individual VMs for different languages. Improving one of these components consequently extends to multiple languages. New optimizations added to the Graal compiler, for example, can be applied to improve the run-time performance of all GraalVM languages.

Moreover, many polyglot VMs use a shared IR to represent different languages based on which language interoperability can be provided. In the .NET framework, for example, all languages are compiled to the Common Intermediate Language, which can then be executed on its CLR [60]. GraalVM languages, on the other hand, are implemented as AST interpreters and when multiple languages are combined, ASTs from these languages are mixed. The Virtual Virtual Machine and Smalltalk/X support multiple bytecode sets to allow the execution and integration of different languages.

What sets polyglot VMs apart from other approaches based on shared IRs is that they provide the infrastructure for language interoperability, not others. This, however, usually means that reference language implementations cannot be re-used and that new implementations are needed instead. Therefore, polyglot VMs are usually complex software systems and expensive to build. To lower these costs, they sometimes provide different means that help to implement languages. As part of the Dynamic Language Runtime, for example, .NET provides re-usable components for the implementation of dynamic languages [61]. GraalVM, on the other hand, provides a language implementation framework including a DSL for implementing AST interpreters [69].

In terms of runtime performance, polyglot VMs have several advantages over other established approaches for language integration. Compared with CFFIs and IPC, they do not rely on external, OS-provided abstractions.

As a consequence, performance overheads imposed by marshaling can be avoided. Objects can be directly passed between languages by reference, which avoids data duplication and with that additional synchronization overheads. Furthermore, recent research has demonstrated that JIT compilers deployed within polyglot VMs can perform performance optimizations across language boundaries, which further improves the performance of language interoperability [5, 58].

By avoiding external abstractions, polyglot VMs also have another advantage over CFFIs and IPC: Based on their internal abstractions, they can provide infrastructures that allow tools to work across their guest languages. The .NET framework, for example, provides a profiling API and cross-language debugging based on its CLR [60]. Recent work on GraalVM has shown that debuggers and other tools can be provided through its language implementation framework [210]. With this, language developers can unlock numerous tools for their language with less effort compared to writing these tools from scratch. Similar to improvements for internal VM components, improvements for tools and new tools can be shared across guest languages.

[|]

Polyglot VMs such as GraalVM and .NET already provide many opportunities for developers in practice. Nonetheless, polyglot VMs are still an interesting research topic as their use cases and requirements are not as well understood as those of other language integration techniques. They need to stay general enough to accommodate different types of programming languages. At the same time, they want to make as much code as possible re-usable, which requires appropriate concepts for language interoperability.

Overall, we think that polyglot VMs have the potential to change the way developers approach and apply polyglot programming. So far, it has often been either a necessity or an alternative, for example, to avoid porting code from one language to another. The easier it is to combine the strengths, libraries, and frameworks of different programming languages, the more likely developers are going to use polyglot programming to build complex applications.

Appropriate tools are key to make polyglot programming more approachable for developers. Polyglot VMs have already demonstrated that debuggers and other common programming tools can be provided across multiple languages. While this is a good starting point, we believe that some tools also need to be extended and new tools need to be built that support polyglot programming. For this reason, this work presents an approach for an exploratory tool-building platform. On the one hand, such a platform helps tool developers to explore tooling ideas that support application developers in building

polyglot applications in new ways. On the other hand, exploratory programming also helps language and runtime developers to better understand how polyglot VMs can or should be evolved.

Summary In some programming domains, such as when building database applications, it is already common to apply polyglot programming. Developers also use it for other purposes such as to re-use existing code across languages as an alternative to porting code.

Established language integration techniques such as C-based foreign function interfaces or inter-process communication are usually based on low-level abstractions provided by the operating system. These abstractions come with limitations, for example, in terms of run-time performance and tool support. Languages built on top of others can avoid some of these limitations but are often limited to only a pair of languages. Consequently, polyglot programming has been limited in use so far, often involves tradeoffs, and is sometimes applied unknowingly or only out of necessity.

Polyglot VMs, on the other hand, are designed to support multiple languages and provide a new level of polyglot programming that does not compromise on run-time performance or tool support. GraalVM and .NET already provide many of these opportunities for developers in practice. With appropriate programming tools and good application examples, we believe polyglot VMs can change the way developers approach and apply polyglot programming.

Part II.

Background

3. Introduction to Programming Languages, VMs, and Tools

This chapter provides a general introduction to programming languages, virtual machines, programming tools including tools for exploratory programming, and different developer roles and responsibilities.

3.1. Programming Languages

Programming languages allow developers to write programs that can run on computers. In the context of polyglot programming, it makes sense to look at four general differences of programming languages.

The first difference is how languages are executed: An implementation of a language either *compiles* or *interprets* code, using a *compiler* or an *interpreter* respectively. When source code is compiled, it is translated to machine code before execution. An interpreter, on the other hand, translates code during execution. While a language is typically designed for one of the two execution modes, it is possible to interpret languages that are typically compiled and vice versa.

The second notable difference is the type system of a programming language [150]: *Statically-typed* programming languages usually perform *static* type checking. Languages that perform *dynamic* type checking are referred to as *dynamically-typed* programming languages. Some languages check types both statically and dynamically. Types are checked statically based on the source code, so type safety is guaranteed to some extent before the program is executed. Dynamic type checking, on the other hand, is performed at run-time. Type-checking errors that occur at run-time usually lead to program errors. Some languages allow developers to recover from such errors, others might not and crash instead.

The third difference is the level of abstraction that languages provide. *Low-level* programming languages are very close to the hardware and therefore provide only little to no abstraction from the underlying architecture. While it is easier to write code that makes good use of a system in a low-level language, the code is usually not portable and missing abstractions require additional work on the side of the developer. *High-level* programming languages abstract

from the underlying architecture and provide developers with abstractions to write portable code and in a more expressive and concise way. On the other hand, these additional abstractions might cause performance overheads. A lot of work has been done to reduce such performance overheads, for example with JIT compilers using sophisticated optimization techniques.

The fourth difference is in the programming paradigms that languages support [49, p. 137]. *Imperative* programming languages allow developers to write programs by providing step-by-step instructions to be executed by a computer. Common imperative paradigms are procedural programming, in which procedures are composed to build programs, and object-oriented programming (OOP), which is based on the idea of objects that have state, behavior, and identity. In *declarative* programming languages, on the other hand, developers declare properties of the result that should be computed, not instructions for how it can be computed. Examples of declarative paradigms are functional programming and constraint programming. In the former, developers can specify a series of functions that should be applied to compute a result. The latter allows them to declare constraints that should be solved by the computer. Many languages support more than one programming paradigm and sometimes even combine imperative and declarative elements. They are thus referred to as *multi-paradigm* programming languages.

3.2. Virtual Machines

Virtual machines are software systems that, from a high-level perspective, provide emulation as an abstraction for other software that runs on top of them. In general, there are two kinds of VMs: *system* and *process* VMs [179, p. 23]. The former provides emulation of real hardware, allowing the execution of entire operating systems. Process VMs, on the other hand, usually run on top of an OS and allow the execution of application code. These VMs usually consist of different components that interact with each other, such as an interpreter, a garbage collector, and a JIT compiler. In the following, whenever use the term *virtual machine*, we refer to process VMs.

One of the main components of a VM is an interpreter [179, pp. 29–32]. An interpreter can execute source code directly, as opposed to compiling all sources to machine code prior to execution using a compiler. For this, interpreters usually process source code with a parser first. The result could, for example, be an abstract syntax tree that can be directly executed or another intermediate representation, such as bytecode, which can then be executed by the interpreter. AST interpreters usually evaluate an AST from the leaf nodes to the root node, each of them encapsulating the semantics and behavior of the corresponding language for the corresponding program.

Bytecode interpreters, on the other hand, commonly use some sort of bytecode loop that fetches, decodes, and then executes pre-compiled bytecode for the program in question. While interpreters themselves are usually implemented for specific hardware platforms and OSs, they allow application code to be written in a platform-agnostic way, hence making it portable. This makes it easier for developers to target different platforms and operating systems.

Virtual machines also provide automatic memory management and garbage collection [80]. Allocating and freeing up memory manually is an additional burden for developers. More importantly, it is prone to errors. VMs usually deploy garbage collectors to automate this process. Garbage collectors frequently scan and free the heap from obsolete chunks of used memory. For this, many different algorithms and heuristics exist. Nonetheless, many VMs allow comprehensive configuration of their GC, some VMs even allow the user to choose from a list of different GCs and GC algorithms. While automatic memory management is a common feature of high-level languages, it does not always avoid memory-related issues such as memory leaks.

In addition, some virtual machines also come with a JIT compiler to accelerate the execution of application code at run-time [179, pp. 147–218]. In a first step, they often profile a running application to determine which parts of the codebase are frequently used and could benefit from dynamic optimizations. Code that is marked as hot is usually then further analyzed by the JIT compiler. Oftentimes, they perform many different optimizations, such as inlining or dead code elimination, as part of different phases. Finally, they produce optimized machine code that can be executed instead of interpreting the corresponding code as usual. Some optimizations make certain assumptions, for example about types or ranges of specific values. If such an assumption is no longer valid, optimized machine code must be thrown away and may be replaced with a more general version. For this purpose, optimized code often contains special guard instructions that allow the VM to fall back to the interpreter from running optimized code.

While most VMs are designed for a single programming language, there are different approaches that allow the execution of different languages on top of the same VM. A common way is to use a shared IR across different languages. Some languages, for example, can be compiled to bytecode of other languages. Polyglot VMs can also be built in different ways: They can use different parsers that produce the same kind of ASTs as shared IR. Other approaches compose multiple interpreters within the same polyglot VM. These and other approaches to language integration are discussed in more detail in Section 2.2.

3.3. Programming Tools

Programming tools are applications designed to support developers throughout the software development process. For code reading and writing, developers often use interactive code editors, such as EMACS [189] or VS Code [113]. These editors usually operate on files and provide syntax highlighting and help developers to navigate through their codebases. Version control systems, on the other hand, allow fine-grained versioning of source code based on files. And build tools help to automate the build process of applications. Furthermore, there are numerous tools based on static program analysis that help with the code writing activity: With linters, developers can find syntactical and stylistic problems in their code. Other static tools help, for example, to detect duplicated code or security issues.

While static tools can help to find many errors at development-time, some programming errors manifest themselves only at run-time and usually require dynamic tools to be understood and caught. Common examples of this are performance problems. Performance analysis tools such as tracing and sampling profilers, for example, help developers to understand where most of the time is spent in their applications. Debuggers, on the other hand, support developers to identify and fix unexpected or erroneous run-time state within their applications. Read-eval-print-loops (REPLs) allow developers to evaluate expressions, for example, to try out code snippets and to explore APIs. With code coverage tools, developers can check which parts of their codebase are covered by tests.

Some programming tools are designed for one specific programming language or virtual machine, for example, when they provide language-specific feedback on code style or run-time state. Others support multiple languages so that developers can use the same set of tools for developing code in different languages. These general-purpose tools, however, usually assume that only one language is used at a time. Integrated development environments increase developer productivity further by bundling different tools, such as code editors and debuggers, in one environment. Similar to individual tools, IDEs can also be language-specific. General-purpose IDEs, such as Eclipse [33] or Visual Studio [112], on the other hand, provide comprehensive means for software development across a wide variety of mainstream programming languages.

Tools for Exploratory Programming

The goal of exploratory programming [172, 175, 203] is to help developers to understand the dynamic behavior of the programs that they build. For

this, exploratory tools rely on and visualize dynamic run-time data and therefore operate at run-time and can co-exist with other applications. The two essential features are interactive code evaluation and object inspection. More features for exploratory programming may, for example, be provided by the underlying programming language of the programming system that provides the tools.

Interactive Evaluation of Code The most basic features of exploratory programming are based on the ability to interactively evaluate expressions within a REPL or another running system, such a self-sustaining programming system or a testing environment, and to see the result somehow. These basic features are usually provided through different commands that developers can trigger individually, such as:

- an *evaluate expression* command that evaluates selected code and does nothing else,
- a *display expression* command that evaluates selected code and displays the result,
- an *inspect expression* command that evaluates selected code and opens the result in an inspection tool, and
- a *debug expression* command that evaluates selected code in an interactive debugger. This is a good example of how exploratory tools can be connected with tools for debugging and other purposes.

In many programming systems that support exploratory programming, these commands are globally accessible. Wherever text can be written, it is possible to trigger these commands for a specific selection or the current line. In addition, they often provide a dedicated workspace tool that can be used as a playground for more complex tasks. For this, these workspaces provide additional features, such as syntax highlighting, a dedicated scope specific to each instance of the tool, and the ability to save code in external files.

Inspection Tools Exploratory programming systems further provide tools for the inspection of objects. These tools visualize the internal structure of objects in different ways, for example with tree lists, which are often sufficient to explore smaller object graphs. For this, they use reflection to list variables as well as variable parts and to access meta-objects such as classes. Moreover, they provide the ability to jump to other tools that, for example, reveal the interface of the inspected object. In addition to commands for interactive code evaluation, they also allow developers to send individual messages to the object under inspection.

Language Features Some systems provide additional features that are useful, but not limited to exploratory programming. Some languages, for example, provide the ability to enumerate all instances for a given class. This mechanism can be a powerful meta-programming tool for applications. It can also be very useful for exploratory programming as it allows developers to find all other versions of a particular instance. In addition to this, languages may provide special means for extending software at run-time, which also aids exploration. An example of this is the ability to change class schemes at run-time. Whenever an instance variable, for example, is added or deleted, all instances of the corresponding class may need to be migrated to reflect this change. Another language feature that can be useful for exploratory programming is a facility for swapping the identities of two objects. With this, it is easy to replace a means for exploration with a different version or an entirely different artifact throughout the entire system.

Figure 3.1 shows a screenshot of the basic tools for exploratory programming in the Squeak/Smalltalk 1.13u programming environment. The "Workspace" tool allows interactive evaluation of Smalltalk expressions. On the right, there is a "Wandering Letters" demo application that was launched by evaluating the second line of the workspace. The demo, written by Ted Kaehler, visualizes an algorithm for the computation of line breaks. Two object inspectors are opened on this demo: one on a "Folder3" object that holds the contents for the demo and one on "SqueakView", the actual view of the application. These two inspectors list the instance variables and their contents and allow the user to evaluate code in the context of the inspected object. Interactive evaluation of code is not limited to the workspace and can be done throughout the entire system using different commands: With *doIts*, selected text can be quickly evaluated. *printIts* do the same and display the evaluation result at the end of the selected text. To open the result in an inspector, the *inspectIt* command can be used. And the *debugIt* command starts a debugger session for the selected text. Furthermore, Squeak/Smalltalk provides support for finding all instances of a given class. Instead of navigation through the object graph of the demo to find the view object, the "SqueakView" inspector was opened through the third line in the workspace using the `allInstances` language feature.

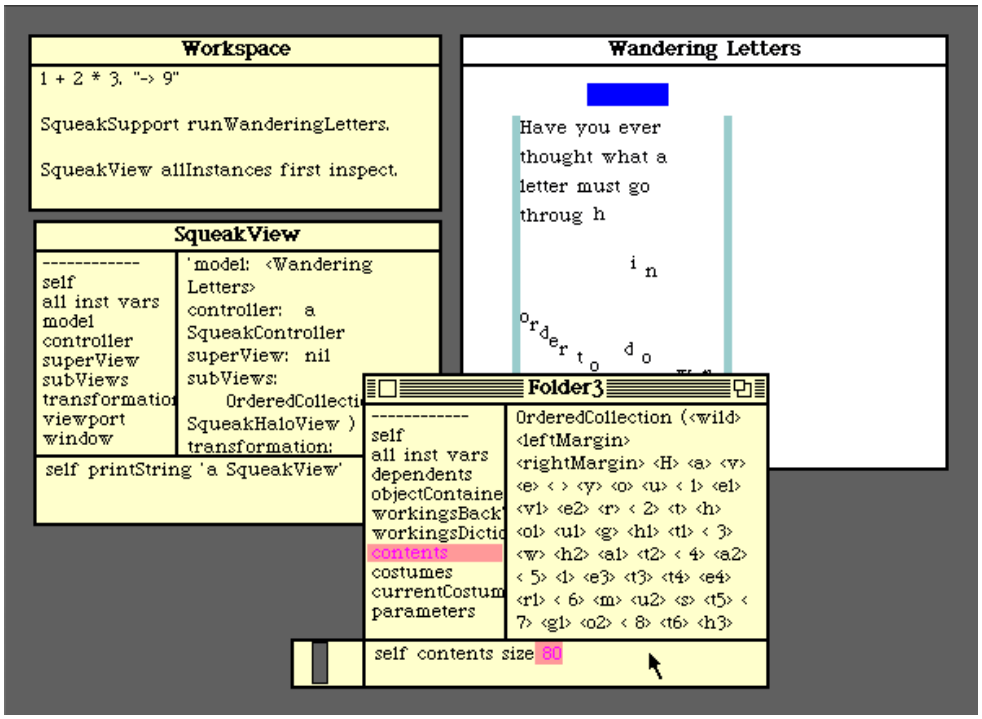


Figure 3.1.: Screenshot of some of the exploratory programming tools in the Squeak/Smalltalk 1.13u programming environment (released in 1996). The "Workspace" tool allows interactive evaluation of code. The second line of the workspace started the "Wandering Letters" demo on the right. The windows labeled "SqueakView" as well as "Folder3" show the object inspector tool. The former is opened on the view object of the running demo, the latter on some user-defined folder object that contains the contents displayed in the demo. In the lower part of the object inspector, code can be evaluated in the context of the inspected object.

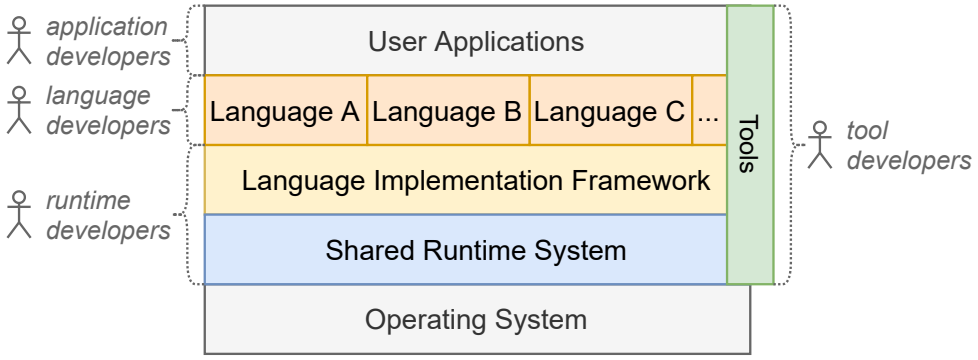


Figure 3.2.: Overview of the different developer roles and their responsibilities within the ecosystem of a polyglot virtual machine.

3.4. Developer Roles and Responsibilities

Figure 3.2 provides an overview of the different developer roles and their responsibilities within the ecosystem of a polyglot VM. In the remainder of this thesis, we use the following names to refer to specific developer roles:

Application developers build and maintain user applications using one or more programming languages and can therefore be seen as the end-users of the ecosystem. The programming tools usually used by application developers are code editors, build systems, and debuggers. They often use and sometimes develop software libraries and frameworks.

Language developers build and maintain implementations of programming languages, for example in the form of compilers or interpreters. Language implementation frameworks allow these developers to use high-level programming languages and to re-use appropriate components.

Tool developers build and maintain tools and IDEs for software development. Sometimes language developers slip into this role and build tools for their language implementations. Tools, however, can also be built and provided by third parties. Tool developers often use tool-building and UI frameworks.

Runtime developers build and maintain runtime systems including components such as JIT compilers or garbage collectors. They may also develop language implementation frameworks that language developers can use for implementing languages for their runtime systems.

Summary There are many different kinds of programming languages: They can be compiled or interpreted, statically-typed or dynamically-typed, high-level or low-level, and support different programming paradigms. Developers must be aware of such differences when applying polyglot programming.

Process VMs allow the execution of application code and can provide portability across different hardware platforms and operating systems. They deploy interpreters, garbage collectors, as well as JIT compilers, and can be built in such a way that they support multiple languages at the same time.

Furthermore, there a numerous programming tools, such as code editors and debuggers, that support developers in building software. Such tools are sometimes bundled in general-purpose IDEs that support a wide variety of different programming languages.

Moreover, tools for exploratory programming help developers to understand the dynamic behavior of the programs that they build by allowing them to interact with their programs at run-time.

In the remainder of this thesis, we distinguish between four kinds of developers: application, language, tool, and runtime developers.

4. GraalVM and Its Infrastructure for Polyglot Programming

GraalVM is a high-performance, polyglot virtual machine developed by Oracle Labs. The project builds on the Java HotSpot Virtual Machine and consists of many different components as shown in Figure 4.1. The central component is the Graal compiler, a versatile JIT compiler written in Java. This compiler supports the execution of programming languages implemented in the Truffle language implementation framework [69]. In addition to re-usable components for implementing languages, Truffle provides several APIs that allow interoperability between different languages and the instrumentation of code. In the context of GraalVM, all languages implemented in Truffle are referred to as *guest languages*, as opposed to the *host language* Java. Furthermore, GraalVM provides various tools for application, language, tool, and runtime developers. In the following, we explain these key components in more detail. Any highlighted implementation detail is taken from the GraalVM 21.2.0 release [29].

4.1. The Graal Compiler

Graal is a modern Java compiler that supports different modes and use cases. Unlike the two default Java compilers of the JVM, C1 and C2, which are written in C++, Graal is written in Java.

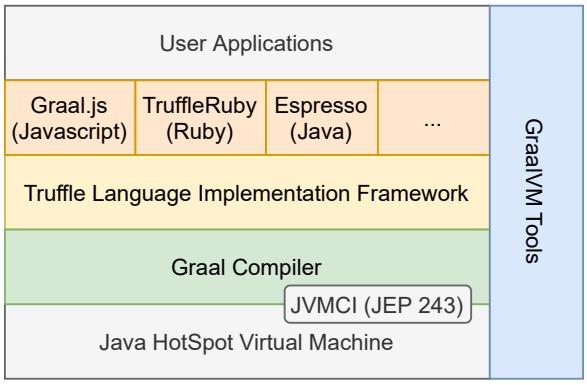


Figure 4.1.: Overview of the GraalVM technology stack.

Internally, the compiler uses a graph-based IR that allows dynamic speculative optimizations [36]. It hooks into a JVM through the JVM Compiler Interface (JEP 243 [30]) and can be used as Java compiler instead of C2 and in conjunction with C1. In this *JVM mode*, Graal performs numerous, state-of-the-art optimizations including aggressive inlining, on-stack replacement, scalar replacement, and stack allocation [114, 188].

In addition, the Graal compiler supports ahead-of-time (AOT) compilation [216]. GraalVM Native Image leverages this *AOT mode* of the compiler and allows the translation of Java applications into native executables. These executables have a low number of dependencies and are independent of the JVM.

Furthermore, the Graal compiler has dedicated support for language implemented in the Truffle framework. In this mode, the Graal compiler performs *partial evaluation* [173, 194], which allows GraalVM to be used as a high-performance, polyglot VM.

4.2. The Truffle Language Implementation Framework

Truffle is GraalVM's language implementation framework and used to implement all languages supported by the VM [221]. It is written in Java and designed for building AST interpreters. Instead of having to implement languages from scratch, Truffle provides re-usable components for language implementations, such as a dynamic object storage model or an infrastructure for managing stack frames. It also provides several tooling APIs, for example an Instrument API for building tools such as debuggers and profilers in a language-agnostic way [210]. Since GraalVM is based on Java and the JVM, Truffle interpreters are also written in Java (or other JVM-based languages such as Kotlin) and can re-use the Java ecosystem including its object model and garbage collectors. Truffle ASTs are used as the common IR of all GraalVM guest languages. Furthermore, Truffle interpreters are normal Java applications that can run on stock JVMs. Only when running on the Graal compiler, however, these interpreters benefit from being partially evaluated, which can significantly improve their run-time performance.

Partially Evaluating Truffle Interpreters The Graal compiler performs partial evaluation to optimize Truffle AST interpreters at run-time [221]. That is, it specializes Truffle ASTs by rewriting AST nodes for the currently running user application. The idea of partially evaluating an interpreter was first discussed by Futamura and is known as the first Futamura projection [48]. The way Graal applies this optimization technique can be described in three phases:

In the first phase, an AST interpreter runs as a normal Java application on top of GraalVM. This is often referred to as *interpreted* mode during which the user application is profiled. Profiling an application in the context of Truffle means collecting run-time information such as concrete values or types, or the control flow branches that were taken. Graal uses different heuristics to identify *hot* methods of the running application. These are, for example, methods that are called thousands of times and that may benefit from being JIT-compiled.

Graal maintains a compilation queue to which hot methods are added at run-time. For each method in the queue, it specializes the Truffle AST representing the method for the collected profiling data. This is the second phase of the specialization process. The output of this partial evaluation process is optimized machine code that can run when the method is called. This machine code is specialized because it only contains code for the profiled characteristics of the user application.

In case control flow leads to a program state not covered by the machine code, for example when the method is called with arguments of new types, Graal reconstructs the corresponding state of the interpreter and falls back to the interpreted mode. This third phase is referred to as *deoptimization*.

At this point, the method re-enters phase one and more run-time information is collected for it in the interpreter. Based on that, the Graal compiler may later decide to re-specialize the method. The resulting machine code will then be more general than the previous version, but it can of course lead to deoptimization again as well.

The assumption behind this optimization technique is that program behavior eventually stabilizes and respecialization is no longer required. This, however, can take a significant amount of time, especially when the user application is large and different parts of the application are used at different times. Running a large test suite is another example of a workload that may benefit from JIT compilation only to a limited extent because tests usually only run once, and the overhead of specialization may not be amortized.

High-Performance Language Integrations Languages implemented in the Truffle framework produce Truffle ASTs, which the Graal compiler can heavily optimize. Every node of such an AST must inherit from the same `Node` class in Truffle. Therefore, Truffle ASTs can be viewed as a common intermediate representation for all GraalVM languages.

Figure 4.2 illustrates what happens when methods of different Truffle languages are mixed. a) and b) of Figure 4.2 show ASTs for two methods written in two different languages. The first method performs string concatenation and appends a unit to a number. The second method converts a value from

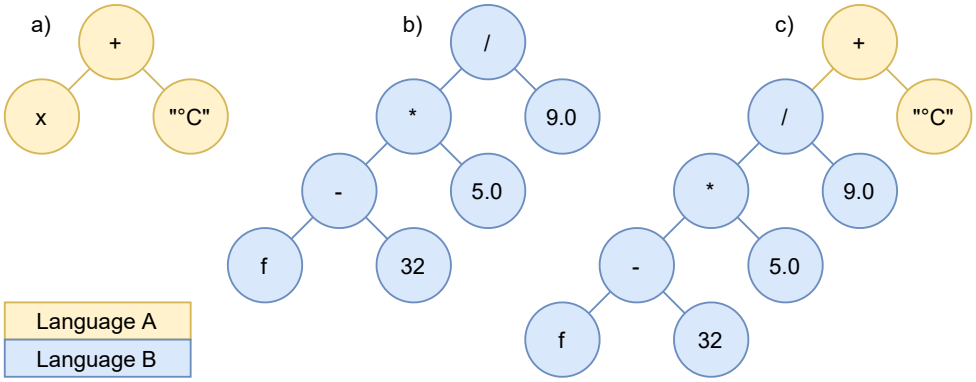


Figure 4.2.: Polyglot Truffle ASTs. a) AST for a simple method performing string concatenation in language A (pseudo-code: `x+"°C"`). b) AST for a method to convert a value from Fahrenheit to Celsius in language B (pseudo-code: `(f-32)*5.0/9.0`). c) Polyglot AST resulting from combining both methods.

Fahrenheit to Celsius. These two methods can be combined to create a new method that accepts a Fahrenheit value and returns a string of the corresponding Celsius value and the unit. The representation of this combination on the AST level is visualized in c): The result is still a Truffle AST but it now contains nodes from two different languages. And since each language implements its semantics within its nodes, the combination will indeed return the expected result.

The origin language of a node is not required for partial evaluation. Consequently, there is no difference between monoglot and polyglot ASTs from the perspective of the compiler. Note that different language semantics may still have different performance characteristics, which is reflected when languages are mixed. Mixing, for example, a language that safely accesses memory (e.g. JavaScript) with one that does not (e.g. C) will lead to code with both safe and unsafe memory accesses. When memory is accessed safely, bounds checks introduce a performance overhead. Consequently, the overall run-time performance of polyglot code can heavily depend on the language in which more memory accesses are performed [58].

Language Interoperability A common representation for code of different languages is important but not sufficient to integrate languages. Objects from different languages must also have a common way to communicate with each other, for example, based on a shared internal representation and an appropriate protocol. In Truffle, objects from guest languages must be represented by host Java objects. Language developers can either use Java

primitive types, Java's `String` type, or custom Java classes that implement the `TruffleObject` [144] interface.

Based on this shared representation for guest objects, the framework provides a dedicated, reflective language interoperability protocol. This protocol is based on message passing and allows language developers to define how their objects interoperate with others. For this, classes that implement `TruffleObject` can export the messages of Truffle's `InteropLibrary` [139]. The interoperability behavior of Java primitive types and the `String` type are predefined by Truffle.

The language interoperability protocol supports several types and traits. Interoperability types, such as `Null`, `Number`, `Exception`, or `Meta-object`, are mutually exclusive. An object can thus only represent one or none of these types. An object may, however, have an arbitrary number of interoperability traits. The *executable* and *instantiable* traits, for example, allow language developers to mark certain objects of their language so that they can be called or instantiated from other languages. The *array elements* trait can be used to expose language objects as array-like objects. At the time of writing, the protocol is still actively extended by the GraalVM community.

Moreover, there must be a way to express the combination of languages from within code and an infrastructure for sharing objects between languages. It is the language developer's responsibility to provide appropriate means for this based on dedicated Truffle APIs and on the level of their language. Official GraalVM languages typically provide a polyglot API through builtins or a dedicated language module. This API usually allows application developers to evaluate foreign code and to share values using Truffle's polyglot bindings object, a dedicated polyglot namespace. The underlying infrastructure for building such an API is provided by `TruffleLanguage.Env` [143], another Truffle-internal API that languages can use to interact with the execution environment.

4.3. GraalVM Languages and Tools

Several programming language interpreters have been implemented in the Truffle framework for the GraalVM. Table 4.1 shows the language implementations that are developed and maintained as part of the GraalVM project as well as several third-party language implementations. From the nine official languages, only Graal.js and Sulong are officially supported. All others are considered experimental, mostly for compatibility and performance reasons. In addition to those, third parties have also created several language implementations. Enso, grCuda, and Yona are, for example, entirely new languages built with Truffle. TruffleSqueak is based on Squeak/Smalltalk and the system

Table 4.1.: List of official and some notable third-party language implementations for the GraalVM.

	Name	Description
Official	Espresso	Meta-circular Java bytecode interpreter.
	FastR	Implementation of GNU R.
	Graal.js	ECMAScript 2020-compliant implementation of JavaScript.
	GraalPython	Implementation of Python 3.
	GraalWasm	Interpreter for WebAssembly.
	SimpleLanguage	Toy language to demonstrate Truffle features.
	Sulong	LLVM bitcode interpreter.
	TRegex	Regular expression engine for other Truffle languages.
	TruffleRuby	Implementation of Ruby.
Third-party	Enso	A visual language for data science.
	grCuda	Truffle-based CUDA integration for GPU programming.
	SOMns	Newspeak implementation for concurrency research.
	TruffleSqueak	Squeak/Smalltalk VM and polyglot programming environment. *(The system presented in this work.)*
	TruffleSOM	SOM Smalltalk implementation.
	Yona	A minimalistic functional programming language.

presented in this work. SOMns and TruffleSOM are related to it in terms of the language, but do not provide a programming environment.

Furthermore, GraalVM also provides tools for different purposes and types of developers, such as the following:

Debugging Protocols GraalVM supports the Chrome DevTools Protocol as well as the Debug Adapter Protocol for its guest languages. This allows developers to choose from a wide range of debugger UI that support one of the two protocols, such as the Chrome debugger or the debugger of VS Code. The implementations of both protocols are based on Truffle's Instrument API, which means that it is not only possible to debug a single guest language but also to debug across multiple guest languages at the same time.

Language Server Protocol GraalVM experimentally supports the LSP, which decouples IDEs from specific languages [192]. This means that developers are free to choose any IDE or code editor with an LSP client. Based on dynamic run-time data, the GraalVM language server can then provide suggestions for code completion and other features supported by the protocol and across all guest languages.

Profiling Command Line Tools GraalVM comes with a set of profiling tools that work across its guest languages: The *CPU sampler* helps to understand in which code most of the execution time is spent. The *CPU tracer* traces through individual statements and accurately counts their number of

invocations in the interpreter or from JIT-compiled code. The *memory tracer* tracks the number of allocations and corresponding source code locations.

MultiLanguage Shell GraalVM provides a MultiLanguage Shell, an interactive REPL that allows the execution of guest language code.

Moreover, GraalVM provides an extension for Visual Studio Code that helps developers to set up and manage GraalVM installations and provides code completion support for its polyglot API. Language and runtime developers can further use GraalVM's Ideal Graph Visualizer to inspect and analyze compiler graphs. Since GraalVM builds on the OpenJDK, VisualVM and other tools for Java can be used to monitor and interact with the runtime.

Summary GraalVM is a state-of-the-art polyglot VM built on top of the Java HotSpot Virtual Machine. Its Graal compiler performs partial evaluation to optimize AST interpreters written in Truffle, GraalVM's language implementation framework. This framework also provides a dedicated protocol for language interoperability.

Several Truffle-based language implementations, such as for Java, JavaScript, Python, or Ruby, are officially maintained as part of the GraalVM project. Others have been created and are maintained by third parties. In addition, the GraalVM ecosystem also provides several programming tools and tool interfaces. Some of these tools and interfaces, such as different profilers and debugging protocols, are implemented in a language-agnostic way and thus work across all GraalVM languages.

Part III.

Exploratory Tool-Building Platforms for Polyglot VMs

5. Bringing Exploratory Programming to Polyglot VMs

Polyglot VMs are complex software systems consisting of many different, intertwined components such as compilers, garbage collectors, language implementation frameworks, as well as several language implementations. They can provide language interoperability on a high level, which allows languages to interact directly with each other and tools to operate across them. This interaction always happens at run-time and is therefore dynamic, even if static languages are involved. Supporting developers in writing code in dynamic programming languages can be challenging for purely static tools. Therefore, we argue that polyglot programming can be better supported with tools based on dynamic run-time data. Self-sustaining programming systems, on the other hand, have demonstrated that it possible to deal with a high level of dynamicity through tools for exploratory programming.

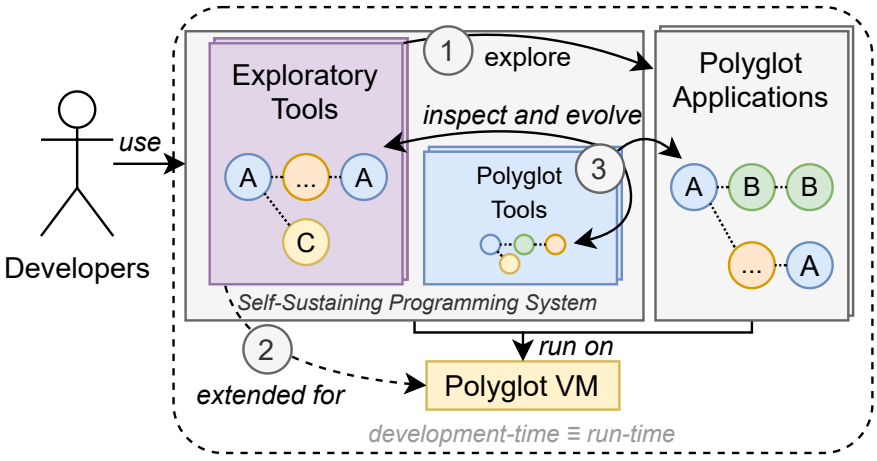


Figure 5.1.: General overview of our approach: ① The tools for exploratory programming of an existing self-sustaining programming system can be reused across all languages of a polyglot VM. ② The exploratory tools are extended so that they provide additional context about language interoperability. ③ Since these tools, just like all other tools of the SSPS, are also just applications, it is possible to evolve them and to build new ones in a polyglot way.

Our approach brings these exploratory tools, tool-building capabilities, and the live programming experience of self-sustaining programming systems to polyglot VMs. Figure 5.1 gives a high-level overview of the approach: ① We show that the tools for exploratory programming of an existing self-sustaining programming system can be re-used across all languages of a polyglot VM. For this to work, an SSPS is integrated in such a way that it runs as a guest language of the polyglot VM. In return, its tools for exploratory programming need little to no changes to be re-usable for other languages. ② While these tools are already useful, they do not incorporate any means to provide context about language interoperability. We, therefore, propose appropriate extensions for these tools that make them polyglot-aware so that they provide additional information about language interoperability, which helps developers to better understand the dynamic runtime behavior within polyglot applications. ③ We demonstrate that polyglot programming can be applied to the entire SSPS when running on a polyglot VM. Existing tools and applications can be extended with other languages and new ones can be built at run-time. This means that we can leverage the extensive tool-building capabilities of self-sustaining programming systems to build new tools for polyglot programming in a polyglot way, allowing us to also gain further insights into polyglot programming by applying it ourselves.

As part of our work, we mostly focus on interpreted, OOP-based programming languages and interoperability between them. As we show in our evaluation in Part V, interoperability between these programming languages already provides a lot of potential for exploration and leads to interesting challenges. Nonetheless, we believe that our approach is not limited to such languages. Since compiled languages can also be interpreted, they can also be combined with other types of languages. GraalVM demonstrates this with its LLVM bitcode interpreter that can interpret C, C++, or Rust code.

In the following, we discuss the potential of exploratory programming in the context of polyglot VMs and explain how the exploratory tools of a self-sustaining programming system can be re-used. Extensions for these tools are presented in Chapter 6. Chapter 7 describes the third aspect of our approach: Polyglot programming can be expanded to our exploratory tool-building platform itself, allowing us to build and evolve polyglot tools and applications at run-time and to learn more about advantages, potential use cases, as well as challenges of polyglot VMs.

5.1. Exploratory Programming for Polyglot VMs

An important activity during the development of any software is to reveal, understand, and specify requirements [51, pp. 391–394]. Only if the require-

ments are sufficiently understood and specified can a software project succeed. This also applies to developing polyglot applications and polyglot VMs. One important way to understand and to find requirements is through exploration [175]. Exploratory programming tools are designed to help developers in this process.

In a survey of exploratory software development, Trenouth [203] describes four principles of exploratory software as follows:

Principle of Execution During the exploration process, the means for exploration must always be "continuously executable". Although this artifact is meant to gain knowledge, it can be entirely different from the software under development. Static representations of the software, on the other hand, fail to provide appropriate means for trying out ideas.

Principle of Extension The means for exploration must always be "easily extendible". Exploratory programming is about experimenting with running software. Developers must be able to modify it quickly. The longer the feedback loop, the harder it is for them to understand the effects of their changes, and to make many changes in quick succession.

Principle of Exploration The means for exploration must always be "conveniently explorable". It must be possible to explore alternatives and to go back to previous versions of the running software. This can be provided through a version control system or copies of the artifact.

Principle of Explanation The means for exploration must always be "usefully explainable". The goal of the exploration process is to allow developers to examine a particular problem or explore a specific design space. At any point in time, they must be able to fully understand their running software.

In the context of polyglot VMs, these four principles have the following meaning. The *Principle of Execution* fits well with our observation that language interoperability as provided by polyglot VMs happens at run-time. While it is possible to write polyglot applications with static tools, developers can only be certain that they work as intended by running them on a polyglot VM. With an artifact that is always running, on the other hand, they can get accurate information about the dynamic behavior of the software artifacts they would like to combine.

The *Principle of Extension* imposes first requirements on polyglot VMs: Running artifacts must be extendible. This means that it must be possible to change existing and to evaluate new code without having to restart the VM. Many dynamic programming languages provide means for dynamic evaluation of code. Furthermore, some of them provide additional means for the extension of application at run-time, for example through incremental compilation and

hot-code reloading. Static languages, however, usually require recompilation of the code that has changed. Since they run interpreted on polyglot VMs, the VM must at least allow the dynamic evaluation of code from static languages at run-time. Another requirement imposed by this principle is independent from specific languages: Run-time performance must be good enough to provide short feedback loops and low response times. If the evaluation of code takes too long, it may be hard to understand how it influences existing behavior or how it adds new behavior.

The *Principle of Exploration* adds further requirements: It must be possible to finely version polyglot applications. Most version control systems operate on the level of files and are therefore not necessarily the best option for versioning running artifacts. Alternatively, the polyglot VM could provide the ability to clone running artifacts. At the very least, it must be possible to re-create different versions of the artifact, for example by allowing code to be re-evaluated.

The *Principle of Explanation* is the most important aspect of exploratory programming and distinguishes it from conventional dynamic tools such as debuggers and profilers. The running artifact must be explainable at any point in time. Tools for exploratory programming and the running artifact must be able to co-exist. Debuggers stop the execution of the artifact and only provide information about it at a particular point in time. While it is possible to step through code, the amount of information they provide is usually too large to be comprehensible over thousands or millions of steps over time. Profilers, on the other hand, run alongside code. Their task is, however, predefined: Measure or trace specific aspects of the execution, such as the time spent per method, until the program terminates. Unlike debuggers and profilers, live object inspectors allow developers to freely explore different aspects of their running applications and are therefore a good example of an exploratory programming tool. The goal of exploratory programming is to enhance the understanding of the software and as this happens, developers might have new questions they want to examine.

Moreover, the last principle has another meaning with regard to polyglot VMs: Everything must be explainable, and different types of developers may want to explore on different levels. Remote tools, such as debuggers that connect to VMs through debugging protocols, are limited to what these protocols allow them to do. Extending such protocols is usually not possible at run-time, which can cause long feedback loops as parts of the polyglot VM need to be rebuilt and restarted after a change to the protocol. We, therefore, argue that it is best for exploratory tools to avoid remote tooling protocols. Instead, they should be integrated into polyglot VMs as closely as possible. Similarly, guest languages may only provide high-level means

for language interoperability. Exploratory tools on top of one of the guest languages of a polyglot VM could therefore be restricted in a similar way and can consequently be less useful for exploration. In addition, the different components of polyglot VMs are usually connected through well-defined APIs. It is good practice to keep the size of public APIs small and to hide as much information as possible. This also makes it easier to evolve them over time, as they provide a small surface for dependencies. Another reason for some APIs to be kept private is because of security concerns. Security mechanisms or best practices such as information hiding, however, can be counterproductive for exploration and therefore are non-goals for us, unless they are one of the aspects under exploration. The more open different components of polyglot VMs are, the better they can be explored.

5.2. Building on Self-Sustaining Programming Systems

If we build tools for exploratory programming from scratch and connect them to the polyglot VM, similar to how remote tools such as debuggers connect to them, our tools are limited to the functionality of the public communication protocol. Another option is to implement these tools on top of an existing guest language. Not all languages, however, provide means for building interactive tools. More importantly, they usually do not allow direct interaction with the language interoperability protocol of the polyglot VM. Our tools would therefore be limited to the functionality provided by the guest language. Instead, we suggest hosting our exploratory tools on a dedicated guest language so that our system is independent and it is up to us to decide what functionalities are made available. Designing and implementing a language from scratch, and then building a tool-building framework and tools for exploratory programming, however, would require a lot of work. To build on previous work, we propose to re-use an existing self-sustaining programming system that already comes with exploratory tools and tool-building capabilities. Another reason for re-using an SSPS is that, by definition, most of its components are fully accessible and can be evolved from within the system. This sets them apart from many other mainstream development environments, which are often hard to extend because they are complex, proprietary, or deploy a restrictive plugin system for extensions.

Figure 5.2 illustrates what this would look like: To host an existing SSPS on top of a polyglot VM, only its VM interface needs to be implemented as a guest language in the corresponding language implementation framework. Its tools, class libraries, compilers, and other parts of the system are built and maintained within itself and can be re-used without further ado. For other languages, on the other hand, libraries, compilers, and other required

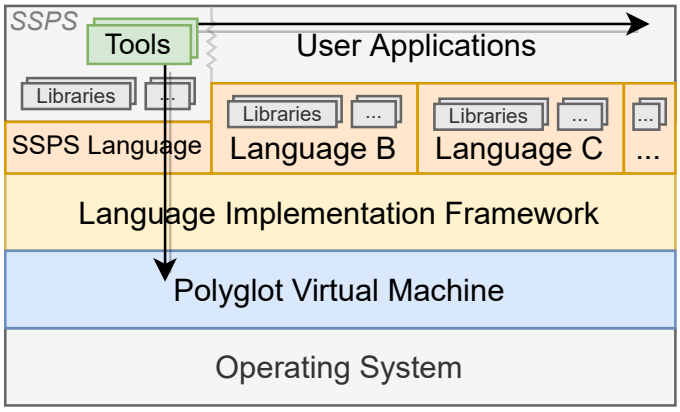


Figure 5.2.: A self-sustaining programming system implemented as a guest language of a polyglot VM. Compared with other guest languages, the language implementation it requires to run is small as most of the components such as class libraries are self-sustained and, therefore, built and maintained within the system. A separate language implementation also makes the programming system independent and allows us to open it up not only to other guest languages but also all the way down to the polyglot virtual machine. As a result, the system can co-exist with user applications, and its tools for exploratory programming can be used to explore them.

components must usually be provided by the corresponding language implementation. Similar to self-sustaining programming systems, portions of this can also be implemented in the language they implement. Unlike in such systems, however, these components are not maintained at run-time and are usually not intended to be changed by application developers. Consequently, the implementation effort required to host an SSPS is lower compared with other language implementations.

Nonetheless, the language implementation needs to properly support all requirements for a particular SSPS. This usually includes an interpreter as well as several language primitives the SSPS relies on. Some of these requirements, however, may be rather uncommon compared with other languages. For one, self-sustaining programming systems usually provide an interactive user interface for both its tools as well as user applications. The language implementation must therefore provide appropriate graphics and windowing facilities. It must also support all language features required by the SSPS. Some of them are related to exploratory programming, such as the capability to enumerate the instances of a particular class that we discussed in the previous chapter. But there may be additional features, such as continuations, interrupt routines, or a mechanism to persist and load snapshots of the system.

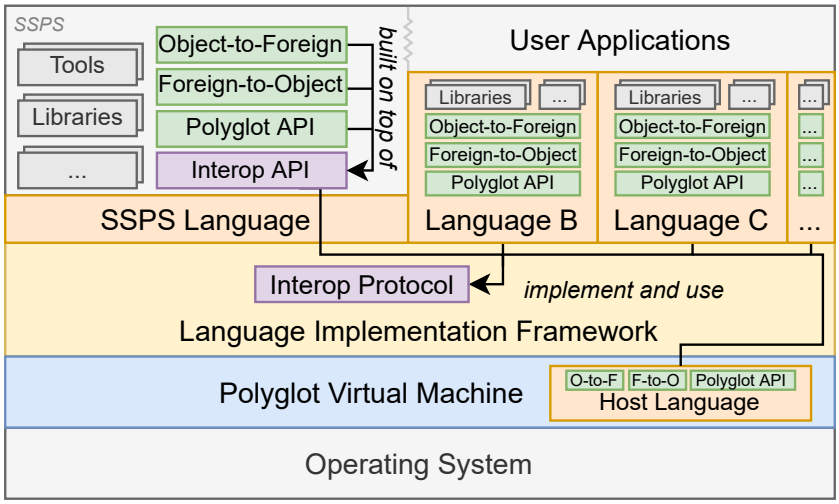


Figure 5.3.: All languages of a polyglot VM need to implement its language interoperability protocol and use it to provide three components: a user-facing polyglot API for evaluating code in different languages, a mapping for their objects going out through the protocol, and a mapping for incoming, foreign object to their objects. Commonly, these three components are implemented on the language level. To enable further exploration of language interoperability, we lift their implementation into the user space of the SSPS.

5.3. Opening the Programming System to Other Languages

The next step towards re-using the tools for exploratory programming is to open up the SSPS to all other languages of the polyglot VM. To provide access to other languages, language developers commonly need to implement and use the language interoperability protocol of the polyglot VM's language implementation framework.

Figure 5.3 shows the three main components, highlighted in green, that language developers need to provide to enable language interoperability:

1. A polyglot API that enables user applications to evaluate code written in other languages. This API is usually provided through a dedicated, global module, class, or set of functions.
2. A mapping from objects of their language to the interoperability protocol. This mapping is used when objects are passed over to and accessed from within other languages.
3. A mapping from foreign objects implementing the protocol to the language's object model. For this, one or more dedicated types are often introduced for representing foreign objects.

These three required components are usually implemented as part of a language implementation. We propose to lift these implementations into the user space of the SSPS, not just because it follows the self-sustaining philosophy of the system, but also because it allows direct and fine-grained access to the protocol. On the one hand, this allows us to build and explore higher-level abstractions and implementation strategies of the interoperability protocol at run-time. On the other hand, direct access to the protocol enables the construction of polyglot-aware tools. It also makes it possible to use exploratory tools on the level of interoperability protocol, allowing us to understand the effect of specific interoperability messages when sent between specific languages. Instead of implementing the interoperability protocol, the language implementation for the SSPS therefore only needs to make it directly accessible from within the system.

Furthermore, language interoperability is not limited to guest languages. As depicted in Figure 5.3, a polyglot VM can also implement these three components to expose its host language through the interoperability protocol. This can allow guest languages not only to evaluate code in that language but also to access the internals of the polyglot VM at run-time, enabling VM introspection capabilities.

5.4. API Requirements for Exploratory Programming

An important goal of integrating a self-sustaining programming system into a polyglot VM is to re-use its tools for exploratory programming. These tools usually consist of different components such as an interactive user interface, event handling, other internal logic providing tool-specific features, or live feedback mechanisms. More importantly, these tools usually access object information entirely through reflective operations of the language they are designed for. This means that all we have to do to make them work for other languages is to route these reflective operations through the protocol for language interoperability. For this, we override appropriate parts of the object protocol of the SSPS's language within the type or the types used to represent foreign objects. As an example, a *display expression* command can now be used to evaluate foreign code through the polyglot API. Through the interoperability protocol, the command can retrieve a textual representation of the result in a language-agnostic way, which it can then display to the user.

For the tools of exploratory programming presented in Section 3.3, we gather the requirements for the language interoperability protocol of a polyglot VM. With this, we can override the reflective operations of the object protocol in our foreign object types so that the exploratory tools can work across other languages without any further modification. In the following,

we list individual API capabilities that the interoperability protocol needs to provide in detail. Note that some of these capabilities could make use of exceptions to signal errors. For simplicity, we let them return boolean values to indicate whether an operation was successful or not.

Basic Capabilities

For the most basic interaction with other languages, only two simple capabilities are required: A means to request the evaluation of code written in a particular language and the ability to retrieve a textual representation for any object that can be displayed to the user.

```
evaluate(String:language,String:code):Object
  Evaluate code written in a specific language and returns the result.
printString(Object:object):String
  Retrieve a textual representation for a given object.
```

Access to Identity and Meta-Objects of Objects

For exploration, it must further be possible to check whether two objects are identical and to access meta-objects of objects. With meta-objects, alternative versions of objects can be identified during exploration and new ones can be created.

```
areIdentical(Object:object1,Object:object2):Bool
  Returns true if both given objects are identical, false otherwise.
getMetaObject(Object:object):MetaObject
  Retrieve the meta-object of an object, such as its class, constructor, or
  prototype.
isInstance(Object:object,MetaObject:metaObject):Bool
  Returns true if a given object is an instance of a particular meta-object,
  false otherwise.
createInstance(MetaObject:metaObject):Object
  Create a new instance of a given meta-object.
```

Access to Properties of Objects

Exploratory tools help developers to understand the internal structure of their objects. For this, the tools must be able to access a list of all enumerable properties and to read and write specific properties of objects. Most languages

distinguish between named properties, such as fields or variables, and numbered properties, such as for arrays or other variable-sized parts. Our API is kept more general to allow any kind of property type. Through `printString`, tools can always obtain a textual representation for any property. Note that some objects may have dynamic or unknown properties, depending on the language, that cannot be listed and are therefore not part of the result of `listProperties`.

```
listProperties(Object:object):Object[]
  List all enumerable properties of a given object.
readProperty(Object:object,Object:property):Object
  Read a specific property from a given object.
writeProperty(Object:object,Object:property,Object:value):Bool
  Write a value to a property of a given object. Returns true if successful,
  false otherwise.
```

Access to Interfaces of Objects

To interact with objects, developers must be able to send messages to a specific object through exploratory tools. In case the interface of an object is unknown, it must be possible to list possible messages. Similar to properties, however, some languages allow objects to understand dynamic messages that cannot be listed and can thus not be reported as part of the result of `listMessages`.

```
listMessages(Object:object):String[]
  List all enumerable messages that can be sent to a given object.
send(Object:object,String:message,Object[]:arguments):Object
  Send a specific message with arguments to a given object.
```

Optional Exploratory Programming Features

In addition, we identified additional features of exploratory programming that polyglot VMs could optionally provide across languages. While these features are not directly required by the exploratory tools, they can be helpful when using these tools.

Objects can potentially have dynamic properties and understand messages dynamically, both of which cannot be listed. In Python, `__getattr__()` and `__getattribute__()` can be overridden to control property access, for example, to return computed values based on property names or a Python `callable` to dispatch messages dynamically. Similarly, Ruby and Smalltalk support the

`method_missing()` and `doesNotUnderstand:` methods respectively, which can both be overridden to handle messages dynamically based on message names and arguments. To make these mechanisms for dynamic behavior more discoverable, we propose a check for each, innumerable properties and innumerable messages. If an object has none, developers know that the property and message lists are complete. The checks otherwise indicate to developers that more information, for example from the implementation or documentation, may be needed to further understand the structure or behavior of an object under inspection.

While the previous requirements allow the creation of alternatives, dedicated support for cloning makes it easier to fork certain objects. To find other versions of an object during exploration, a mechanism for listing all instances of a specific meta-object is helpful. Similarly, an ability to swap the identities of two objects can be helpful during exploration, for example, to trying out different alternatives or for evolving a running artifact without altering object identities.

`hasInnumerableProperties``(Object:object):Bool`
Returns `true` if a given object has innumerable properties that cannot be listed as part of `listProperties`, `false` otherwise.

`understandsInnumerableMessages``(Object:object):Bool`
Returns `true` if a given object understands innumerable messages that cannot be listed as part of `listMessages`, `false` otherwise.

`clone``(Object:object,Bool:shallow):Object`
Create a shallow or deep copy of a given object depending on the `shallow` argument being `true` or `false`.

`findAllInstances``(MetaObject:metaObject):Object[]`
Find all instances for a specific meta-object.

`swap``(Object:object1,Object:object2):Bool`
Swap the identities of object1 and object2. Returns `true` if successful, `false` otherwise.

`[|]`

Note that most of these capabilities are not specific to exploratory tools. Since they are also needed to support language interoperability and other tools such as debuggers, polyglot VMs usually already provide appropriate infrastructures that can be re-used and extended. An `evaluate` hook and a hook to `send` messages, for example, are crucial for integrating languages in general. `MetaObjects` allow languages to implement an `instanceof` operator for other languages. Similarly, some of these capabilities may also be used by other languages to implement reflection and by debuggers and other tools

that perform introspection. A `printString` hook is usually needed in almost all kinds of tools that display values in some form. Additional capabilities for exploration, on the other hand, may be helpful to enhance exploratory programming but are not necessarily required to allow exploration to a reasonable degree. Also note that while information on possible arguments for messages would be useful, we believe the effort to unify keyword arguments, which can also be innumerable, and other types of arguments across languages outweighs the benefits for exploration. Since polyglot VMs are dynamic systems, we instead assume that objects respond to messages sent with incorrect arguments with useful errors that provide information on, for example, their arity or expected argument names.

Summary Language interoperability as provided by polyglot VMs is dynamic and can best be observed at run-time. Therefore, we argue that exploratory programming is a useful practice to understand the requirements of tools for polyglot programming, polyglot applications, and polyglot VMs. Unlike static tools and many conventional dynamic tools such as debuggers and profilers, exploratory tools co-exist with running exploration artifacts and allow developers to understand the dynamic behavior of these artifacts at run-time.

Instead of creating exploratory tools from scratch, we propose to build on an existing self-sustaining programming system. This allows us not only to re-use its exploratory tools but also its tool-building capabilities and live programming experience. This way, we can build and evolve tools and polyglot applications at run-time, which makes for short feedback loops and thus improves productivity. We show how an SSPS can be integrated as a language of a polyglot VM and how it can be opened to other guest languages. We also list the API requirements in detail that a polyglot VM needs to provide to support exploratory programming. Since these requirements overlap in large part with what these VMs already need to provide to support language interoperability and tools such as debuggers, their implementation can usually build on an already existing infrastructure.

An exploratory tool-building platform as proposed by our approach makes it possible to explore ideas and rapidly build tools for polyglot programming that incorporate dynamic run-time data. This addresses the first challenge from Section 1.1 and thus constitutes the first contribution of this work.

6. Extending Exploratory Tools for Polyglot VMs

The exploratory tools of an SSPS commonly use reflection to access information on objects. We have shown that this can be routed through the language interoperability protocol and therefore, these tools need little to no modification to work across the languages of a polyglot VM. However, they sometimes fail to provide enough context about different languages and language interoperability as they were originally designed for one specific language. Since these tools allow exploration of dynamic object state at run-time, they are also further away from code compared with, for example, debuggers that overlay code with run-time state. In this chapter, we propose different extensions that make exploratory tools polyglot-aware and thus more useful to the different developers working with polyglot VMs.

6.1. Revealing Interfaces of Objects

Exploratory tools for object inspection usually focus on the visualization of program state. For exploring the interfaces of objects, they often provide quick access to other tools that are designed for listing, implementing, and extending interfaces. While this works well in a monoglot SSPS, it is hard to provide the same experience in a polyglot environment as languages might have different approaches to how interfaces are defined and managed. Some languages, for example, allow interfaces of particular classes or even particular objects to be extended dynamically, for example through monkey patching. The actual interface of an object at run-time can therefore be hard to determine with static code analysis.

We thus propose to extend exploratory tools so that they also list the interfaces of objects explicitly alongside object state at run-time. This extension is relatively simple and only needs to be implemented once if based on the language interoperability protocol of the polyglot VM. Having access to the lists of object interfaces is particularly helpful when trying to connect different pieces of software. If developers must consult the API documentation of one piece of software written in one language and the implementation of another piece of software written in another, this not only is time-consuming but

also prone to errors. With exploratory tools displaying interfaces at run-time, developers can always see and understand how objects of different languages with different interfaces can interact in polyglot applications, following the Principle of Explanation of exploratory software.

6.2. Providing Context About Languages

By looking at state and interfaces of objects, developers may be able to guess the language of the objects in their polyglot applications. This, however, requires a good understanding of each language and is an additional burden for a developer. In a polyglot programming system, tools can support developers by incorporating language information. We propose to extend the exploratory tools for object inspection with the ability to explicitly display the language of a particular object.

Furthermore, exploratory tools that allow interactive evaluation of code, such as a *display expression* command or a workspace, can be inconvenient to use because developers are forced to use the polyglot API to call out to other languages. This can be avoided and the experience streamlined by extending these tools with the ability to switch between different languages. The language used for a *display expression* command and others could be a global preference for example. Similarly, a workspace tool could hide the polyglot API by letting developers select a particular language. For this, the polyglot VM must provide a list of all languages that it supports. Tools must further be able to pass in a local scope when evaluating code. An object inspection tool, for example, usually binds the inspected object to a keyword such as “this” or “self” in its embedded workspace.

Moreover, developers and tools may want to access the globals of a particular language for different purposes. Developers, for example, may need to explore or modify some global state in a specific language or would like to find a specific capability without knowing anything about the syntax of that language. Tools, on the other hand, can use access to globals to automate certain operations in a language-agnostic way. For example, they can identify newly introduced globals by tracking a list of all global names across code evaluations. By writing new globals from one language into the globals of another, they can automate sharing between languages.

In the following, we list the additional capabilities that a polyglot VM needs to provide to support our proposed extensions.

Basic Capabilities

Tools and developers must be able to determine the origin language of any object at all times. They must also be able to retrieve a list of all languages supported by the polyglot VM at run-time. To allow tools to evaluate code within a certain context, the `evaluate` capability should be extended with the ability to pass in specific local variables that are then available during the evaluation of code.

```
getLanguage(Object:object):String
  Get the language of a given object.
listLanguages():String[]
  List all languages supported by the polyglot VM.
evaluate(String:language,String:code,Map:locals):Object
  Evaluate code written in a specific language with a map of local variable
  names to values and returns the result.
```

Accessing Language Globals

We further propose a simple API that can be used to list, get, and set globals of the languages of a polyglot VM. The API gives developers an additional starting point for exploration and allows tools, for example, to manage and share global state between languages,

```
listGlobals(String:language):String[]
  List the names of all globals for a given language.
getGlobal(String:language,String:name):Object
  Get a specific global from a given language.
setGlobal(String:language,String:name,Object:value):Bool
  Set a particular global of a language to a specific value. Returns true if
  successful, false otherwise.
```

6.3. Incorporating Additional Features of Polyglot VMs

For exploratory programming on the level of polyglot user applications, the previously discussed requirements are sufficient to understand and explore program behavior at run-time. The exploratory tools, however, can also be extended with additional features of the polyglot VM.

The language interoperability protocols of polyglot VMs usually support different data types and traits to enable high-level interaction between lan-

guages. Since such a protocol is directly accessible from within our SSPS, it is possible to extend tools so that they display detailed information on different interoperability properties of objects. As an example, the exploratory tools for object inspection can explicitly list the interoperability types and traits supported by an object. They can also provide views that show how these types and traits are implemented in more detail. On the one hand, this is helpful to language developers because it allows them to understand how their language integrates the protocol for language interoperability. More specifically, they are interested in both how objects from other languages appear in their language and how objects from their language appear within others. On the other hand, runtime developers can further benefit from the exploration of the language interoperability protocol, the language implementation framework, and other facilities from the runtime such as its JIT compiler or its garbage collector. The exploratory tools can be used, for example, to compare implementations of the interoperability protocol from different guest languages, to spot inconsistencies across them, and to help gather new requirements for evolving the protocol. To allow exploration of additional runtime facilities such as data structures managed by JIT compilers, these facilities must only be exposed to the self-sustaining programming system through its language implementation, similar to how the interoperability protocol is exposed. At run-time, existing tools can then be extended and new ones can be built around these facilities for specific purposes. If a JIT compiler uses, for example, a data structure to manage compilation tasks, it is possible to build tools that can monitor, analyze, and even change these tasks while the compiler is running.

Summary In the previous chapter, we have shown how the exploratory tools of an existing self-sustaining programming system can be re-used across the languages of a polyglot VM. Since these tools make use of a uniform set of capabilities provided by all languages, they are no longer language-specific. This, however, also means that they can only provide language-agnostic views.

To make them aware of a polyglot VM, we propose extensions for these exploratory tools and list corresponding API requirements. Object inspection tools should provide information on the interfaces and languages of objects, which helps developers to understand how to combine objects from different languages without having to read code or documentation. Tools such as a workspace or commands for interactive code execution can hide the polyglot API of a polyglot VM to streamline the experience for developers. Moreover, access to the globals of languages is useful for exploration purposes and allows tools, for example, to manage and share state between languages. We argue that our extensions not only make the exploratory tools polyglot-aware, the required API can also be used to build other polyglot-aware tools.

This step toward polyglot-aware tools is in line with the second challenge presented in Section 1.1 and constitutes the second contribution of this work.

7. Expanding Polyglot Programming to the Platform Itself

A self-sustaining programming system hosted as a guest language on top of a polyglot VM can be used for more than exploratory programming. In this chapter, we show how the tool-building capabilities of such a platform can be used to explore tooling ideas for polyglot programming. At the same time, tools themselves can be built in a polyglot way. While this is useful for tool developers, an SSPS is not limited to building tools. Other parts of the system can also make use of polyglot programming as they are all user applications from the perspective of the polyglot VM. Lastly, we illustrate how both language and runtime developers can also benefit from the capabilities of self-sustaining programming systems.

7.1. Building Polyglot Tools for Polyglot Programming

Interactive dynamic tools are usually expensive to build as they consist of many different parts such as a user interface with event handling, internal processing logic, and different inputs and outputs for information. The process of designing tools can be even more expensive. When requirements for a tool are not yet well-understood, exploring different ideas can be time-consuming. Moreover, existing tools that someone else has built are often hard to extend, if they even provide appropriate means for extension. And if they do, they oftentimes impose limitations on what third-party extensions are allowed to do.

In self-sustaining programming systems, on the other hand, everything can be extended, no matter whether it is a tool, an application, a class library, or its compiler. To support tool developers, they usually come with frameworks for tool-building that provide different re-usable components such as for text editing, windowing, or event handling. Furthermore, self-sustaining programming systems often support mechanisms for rapid prototyping, for example, through incremental compilation and hot-code reloading. This way, it is possible to modify and extend a tool while one or more instances of the tool are running. This provides short feedback loops so that tool developers can alter and evolve the behavior of their tools quicker.

Since our exploratory platform has direct access to the language interoperability protocol and other APIs of the polyglot VM, new polyglot-aware tools can be prototyped and built at run-time. For this, it is possible to re-use any component, library, framework, or tool that already exists in the SSPS. By composing two or more existing tools, new tools for different purposes can be created.

Furthermore, the integration of the polyglot API is not only useful for exploration. The API can also be used from within tools. This way it is possible to make use of libraries and frameworks written in other languages in the context of tools. This can be useful, for example, when a tool should support different file formats. If one of the guest languages of the polyglot VM comes with a library for reading and writing a specific format, it can be used within the tool. Similarly, it is common that interactive tools provide visualizations of data, such as charts, graphs, and plots. With polyglot programming, tool developers can re-use visualization libraries from different languages within their tools. Also, the existing tools for code editing of an SSPS usually only support syntax highlighting for the language they are designed for. Through the polyglot API, these tools can be easily extended with libraries for syntax highlighting that support many different languages.

Building tools for polyglot programming in a polyglot way allows tool developers to re-use more software. At the same time, it also provides new data points allowing us to learn more about programming with polyglot VMs. As we show in Section 13.2, the insights gained through an exploration platform for polyglot VMs are not limited to the system and can also be applied in other programming systems. This means that our platform can also be used for exploration and prototyping when building other systems.

7.2. Building Polyglot Applications at Run-Time

Apart from tools, self-sustaining programming systems can, of course, also be used by developers to build all kinds of other applications. From the perspective of a polyglot VM, everything that is part of the SSPS is a user application. And since development happens at run-time, our platform can be used by application developers to build polyglot applications at run-time, at least to some extent. The language of an SSPS often supports mechanisms to incrementally build applications and to recover from run-time errors.

These mechanisms, however, cannot always be made available to other guest languages of the polyglot VM. This is also the case for the snapshotting mechanism of an SSPS: While snapshotting of our platform must be properly supported to allow its evolution, it is hard to extend this capability to other guest languages. Most programming languages do not support taking and

loading arbitrary snapshots of their object spaces. Nonetheless, other guest languages could make use of the snapshotting mechanisms of the SSPS's language. For this, they only need to manage the information they want to persist in data structures of the SSPS's language and they need to know how to take and restore snapshots.

This also has a consequence for applications directly built in our platform: Snapshots of the platform can be taken at any point in time and do not persist objects from other guest languages. Therefore, these objects are no longer available after resuming from a snapshot. From the perspective of an application written in the SSPS, it appears as if they suddenly vanished. As most languages have no general support for persistent object memory, we leave cross-language persistence for future work. Instead, applications in our SSPS must always be able to re-create objects from other languages. We found that in many cases, the lazy-initialization pattern [50, p. 112] can help with this.

7.3. Exploring the Internals of Polyglot VMs

Although our platform is designed for exploratory programming and tool-building in the context of polyglot VMs, it can also be useful for language developers and the developers of the runtime.

Language developers can, for example, use our platform to understand interoperability of their language with others in more detail. The exploratory tools allow them to inspect how objects of their language implement the language interoperability protocol of the polyglot VM at run-time. At the same time, they can see how other languages implement this protocol and how objects of other languages appear within their language. Furthermore, they can explore different implementation strategies on the basis of the platform's language. Since we propose to implement the interoperability protocol for both outgoing and incoming objects within the SSPS, the implementation can also be changed at run-time.

If the host language of the polyglot VM is supported through the language interoperability protocol, as previously depicted in Figure 5.3, it is possible to access and introspect internals of language implementations, the language implementation framework, and the VM itself from within our platform. This allows language developers to explore their language implementations at run-time. For example, they can use our platform to statistically analyze certain characteristics of their language that are only visible at run-time. Or they could observe how effective a particular optimization is that they have built into their language implementation. For this, they are not limited to the

exploratory tools. They can, of course, also build their own dynamic tools for their language implementation.

The same applies to runtime developers. They can explore specific components, such as the garbage collector or a JIT compiler, of the runtime at run-time with the exploratory tools. Since these components can be quite complicated, custom dynamic tools can be built for them in our platform. JIT compilers, for example, often perform powerful but complex performance optimizations. Our platform can, therefore, help them to better understand internal state, such as optimization heuristics, and identify issues and potential for improvements.

The only part of our platform that is not self-sustained is its language implementation. Therefore, it is not possible to change it in the same way the rest of the system can be changed at run-time. As discussed in Section 5.2, the language implementation is relatively small, not just compared with our platform but also with implementations of other languages. From the perspective of a polyglot VM, most of our platform is built on the level of user applications, which can provide valuable feedback for language and runtime developers. On the one hand, the programming system itself can produce realistic and non-trivial workloads that are typical for IDEs or UI applications, as we demonstrate in Section 11.2. This can be useful to assess how well the polyglot VM supports such workloads, something that, for example, micro-benchmarks usually do not provide. On the other hand, their small language implementation allows for additional experimentation with different strategies to implement languages in the language implementation framework. This way, for example, new framework components or performance optimizations can be tested and evaluated with lower effort. In other language implementations, this would require more work because they are more complex and, therefore, less flexible.

Summary A platform based on our approach can be used for more than exploratory programming and tool-building. Since the platform is integrated as a guest language of a polyglot VM, it is possible to apply polyglot programming in different ways:

Tool developers can apply polyglot programming to provide, for example, support for different file formats, visualizations, syntax highlighting, or other features in their tools. Application developers can use the language of the SSPS to glue together code from different languages, which enables the construction of polyglot applications at run-time. Language and runtime developers can explore language implementations and the internals of their polyglot VMs at run-time through VM introspection enabled by interoperability with the host language. It is also possible to apply polyglot programming to extend the self-sustaining programming system itself.

By applying polyglot programming in these different ways, we can gain further insights into advantages, potential use cases, as well as challenges of polyglot VMs and discover new requirements based on practical experiences. This helps to tackle the third challenge discussed in Section 1.1 and is the third contribution of our work.

Part IV.

Implementation for the GraalVM

8. Integrating Squeak/Smalltalk Into GraalVM

In this chapter, we present TruffleSqueak, an implementation of our approach. TruffleSqueak is based on Squeak/Smalltalk and an exploratory tool-building platform for the GraalVM. It is open-source, available on GitHub [183], and consists of three main components: 1) an implementation of the VM interface for Squeak/Smalltalk in GraalVM's Truffle framework, 2) a VM-level plugin that grants access to other languages and the underlying runtime system, and 3) Squeak/Smalltalk code that implements and integrates GraalVM's language interoperability protocol allowing the exploratory tools of Squeak/Smalltalk to operate across GraalVM languages. In the following, we highlight some of TruffleSqueak's implementation details.

8.1. Building on Squeak/Smalltalk

Squeak/Smalltalk [70] is an open-source Smalltalk implementation and a direct descendant of the Smalltalk-80 system and its specification [53]. It was created by Alan Kay and Dan Ingalls, who were involved in the original design and development of Smalltalk-80 at the Xerox PARC Learning Research Group. The Smalltalk language is an object-oriented, reflective, dynamically typed, and interpreted programming language. What sets Smalltalk apart from many mainstream programming languages are its self-sustaining programming system as well as the infrastructure that enables it, such as its *become* mechanism or its support for persistent object memory. Smalltalk and hence Squeak/Smalltalk provide different tools for exploratory programming and means for rapid prototyping [172]. In addition, its implementation of the Morphic UI framework [99, 100] as well as its `ToolBuilder` and other tool-building facilities allow Squeak/Smalltalk to be used as a sophisticated tool-building platform [197].

The virtual machine interface that Squeak/Smalltalk requires to run is well-documented. There are different VMs that demonstrate how this interface can be implemented: The OpenSmalltalkVM [73], the reference VM for Squeak/Smalltalk, is written in a subset of Smalltalk that translates to C, SqueakJS [47]

and RSqueak/VM [41] are alternative implementations written in JavaScript and Python respectively.

The first step to use Squeak/Smalltalk as an exploratory tool-building platform for GraalVM is to implement this VM interface in Truffle, GraalVM's Java-based language implementation framework. For this, all Smalltalk objects from a Squeak/Smalltalk snapshot, a so-called *image*, must be represented by an appropriate Java object. Apart from a reader for Squeak/Smalltalk images, our language implementation also needs to provide an interpreter for Squeak/Smalltalk bytecode as well as a list of required primitives. Their implementations are now explained in more detail.

Creating a Squeak/Smalltalk Bytecode Interpreter in Truffle

The original Smalltalk-80 specification includes a well-defined bytecode set. GraalVM's Truffle framework, on the other hand, is designed for building AST interpreters. The first challenge to support Squeak/Smalltalk on the GraalVM thus is to find a way to implement a bytecode interpreter in Truffle. Squeak/Smalltalk makes assumptions about certain bytecodes and their execution, so correctly implementing a bytecode interpreter is important for compatibility. This challenge, however, is not specific to Smalltalk. Other languages such as Java are also bytecode-based and need a bytecode interpreter for compatibility reasons. This is also the case for intermediate representations such as LLVM bitcode or WebAssembly that allow the execution of compiled languages such as C, C++, or Rust.

In TruffleSqueak, there is a dedicated Truffle node for each Smalltalk bytecode. Consequently, the bytecode of a method is transformed into an almost linear AST. Each AST node either has one successor node for the next bytecode, or two in case of a conditional jump bytecode. In the latter case, the node points to its direct successor as well as to either another successor or a predecessor if it is a back jump.

Figure 8.1 shows bytecode (blue) for an example method: Bytecode #1 and #3 are simple bytecodes with a direct successor. Bytecode #2 is a conditional jump that only jumps to bytecode #5 if and only if some condition is met. Otherwise, bytecode #3 is executed. Bytecode #4 is an unconditional jump back to bytecode #2, while bytecode #5 is a return bytecode that signals that the execution of the corresponding method has been completed. This method may therefore be incrementing some value in a loop until some condition is met. This is a common scenario when searching for the index of a specific character in a string for example. The transformation of bytecodes into Truffle AST nodes (green) results in an AST with appropriate edges.

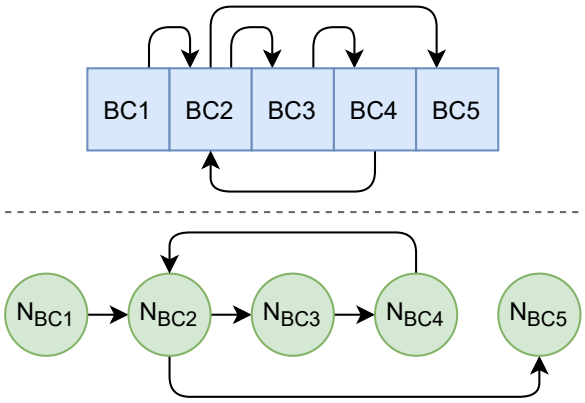


Figure 8.1.: Bytecode (blue) for an example method transformed into a corresponding AST (green).

While the transformation is straightforward, the Graal compiler fails to produce efficient machine code for these special types of ASTs. For this reason, the Truffle framework provides several hints that a language implementation needs to use to help the compiler, for example, to detect loops in application code. Providing these hints in the right way, however, can be challenging and increases the complexity of the bytecode interpreter loop (see Appendix A). On the other hand, similar runtime performance compared with standard Truffle AST interpreters can be achieved, as we show in Part V.

In addition to a bytecode interpreter, some primitives, for example for arithmetic and input-output operations, need to be implemented to be able to run Squeak/Smalltalk. These are also well-documented. While there are hundreds of supported primitives, not all of them are required. Most optional primitives come with appropriate fallback code that provides the same functionality but may be less efficient. The overall strategy is therefore simple: start with implementing all required primitives, then implement optional primitives for better performance and additional features.

A bytecode interpreter with required primitives is able to run the Squeak/Smalltalk compiler. With this, it is possible to evaluate arbitrary Smalltalk expressions. From this point on, we can run the SUnit tests of Squeak/Smalltalk to understand which parts of the Smalltalk system are working and which are incorrect or not yet implemented. For good practice, we set up a continuous integration pipeline that runs all SUnit tests and reports the results every time we push new commits. This feedback cycle helps us to keep track of our progress and to detect regressions early.

Supporting Smalltalk Language Features

A VM for Squeak/Smalltalk needs to provide some powerful, yet uncommon mechanisms for hosting the self-sustaining programming system. This in turn allows many Smalltalk language features, such as exception handling or a mechanism for schema migrations, to be implemented in Smalltalk itself.

`Context` Objects Squeak/Smalltalk provides reification of method and block activations through first-class `Context` objects. To access the `Context` object from within a method, Smalltalk provides a dedicated pseudo-variable called `thisContext`. Since these objects are first-class, they can be accessed and modified just like any other Smalltalk object.

While modifying the sender of a context appears to be a dangerous operation, it is frequently used in Squeak/Smalltalk to modify the control flow. Exception handling is, for example, based on this: If an exception is thrown, the sender chain is traversed until an appropriate exception handler is found. By changing the sender of the current context to the exception-handling context, all intermediate sender contexts are removed and control flow continues executing the exception handler.

The Truffle framework provides a `Frame` infrastructure for managing activation records in guest languages. Since the allocation of such frame objects per activation is expensive, the Graal compiler aggressively tries to avoid allocations on the heap. If possible, it only allocates the required parts of a language-level frame on the stack. In most cases, it can even avoid the allocation of a frame entirely. Language developers can mark a Truffle frame for materialization, which instructs the compiler that allocation on the heap is required for a particular frame.

TruffleSqueak makes use of Truffle frames and tries to avoid materialization of them as much as possible. Allocation on the heap is only required in few cases, for example when a process switch is triggered. Other common cases are non-local returns: In order to allow blocks to trigger an early return within a method, the corresponding block closure needs to be aware of the target context. While marking such target contexts for materialization cannot be avoided, the Graal compiler often decides to inline blocks as they tend to be small. When blocks are inlined, the Graal compiler can detect that the context object does not escape the compilation unit of the method. Therefore, it is again able to allocate only the required parts of the frame or avoid its allocation entirely.

`allInstances` Squeak/Smalltalk allows the lookup of all instances for a specific class object through the `allInstances` method, which makes use of

a corresponding primitive. While the object memory can easily be scanned for specific instances by the GC in the OpenSmalltalkVM, it is a challenge to mimic this behavior correctly in Truffle. One reason is that Java itself does not provide any APIs for interacting with the garbage collector apart from `System.gc()`, which only suggests to run the GC. The same holds true for Truffle: Although it makes some runtime-specific APIs accessible to language developers, there are none for interacting with the GC.

In TruffleSqueak, we thus need to walk over all objects in the heap manually to find all instances of a class, starting from Squeak/Smalltalk's special objects array, an array of objects for the VM that we can use as a GC root. While this approach does not provide the same performance that a GC could achieve, it works well enough in TruffleSqueak. Nonetheless, there are some performance optimizations for `allInstances` in TruffleSqueak: To avoid the allocation of large sets to track seen objects, for example, all objects in TruffleSqueak come with a field to store a marking flag. This is a common GC strategy, only that the information is usually managed within an object's header, not within a field of the object. Another optimization is to avoid marking context objects for materialization by traversing stack frames with read-only access.

`become:` and `becomeForward:` A well-known language feature available in Smalltalk is a mechanism to swap object pointers. With `become:`, all pointers to the receiver object are changed to point to the argument object, and vice-versa. Since this mechanism only needs to support to swap pointers of objects of similar or identical classes, these objects are represented by instances of the same Java class from TruffleSqueak's object model. Therefore, the become mechanism can be implemented by simply swapping the contents of all fields of the two objects in question.

Implementing `becomeForward:`, however, requires additional work. This mechanism works in a similar way, except that only all pointers to the receiver are changed to point to the argument and not the other way around. In this case, the contents of all fields cannot be simply swapped. Instead, TruffleSqueak needs to walk all objects in the heap, similar to how it implements `allInstances`, and update all pointers to the receiver. For this, the implementation of `becomeForward:` can make use of the same optimizations that TruffleSqueak uses for `allInstances`.

Image Snapshots Another powerful but uncommon feature of Smalltalk is its support for image snapshots. For this, the OpenSmalltalkVM can simply dump its allocated object memory into a file. When an image is opened, it loads the file back into memory.

All objects and data of Truffle languages need to be represented by Java objects, and TruffleSqueak is not an exception. Similar to the rest of the Smalltalk VM, the image format is well-documented. TruffleSqueak, therefore, knows how to read the image header and what kind of Java object it needs to allocate for each Smalltalk object. For saving image snapshots, TruffleSqueak again makes use of its `allInstances` infrastructure: It manually walks over all objects in memory and writes out each object in the corresponding format. As a result, TruffleSqueak is fully compatible with the original image format and can therefore load and save images that can then again be opened and saved by the OpenSmalltalkVM and other compatible VMs for Squeak/Smalltalk.

Supporting the Squeak/Smalltalk Programming System

To support the Squeak/Smalltalk programming system, a VM not only needs to be able to interpret bytecode correctly and provide primitives that enable language features. It also needs to support two plugins that provide dedicated drawing primitives: The BitBlt plugin is used for various drawing operations and Balloon for rendering TrueType fonts. Both are traditionally implemented in Slang, a subset of Smalltalk that translates to C, and compiled into the binary of an OpenSmalltalkVM. While it is possible to interpret their original Slang implementations with a simulation infrastructure [73], we manually port the C version of both plugins to Java instead. The main reason for this is that the performance of the two plugins directly influences the responsiveness of the user interface. Simulating the plugins would significantly increase the time until the UI is usable as the simulator and the Slang code of the plugins need to be JIT-compiled first to provide reasonable performance.

Displaying the programming system is, however, only a matter of rendering the display buffer maintained by Squeak/Smalltalk. This display buffer is registered during startup in a primitive. In TruffleSqueak, this primitive uses Java's AWT and Swing to create a window that renders the buffer. Another primitive is used by Squeak/Smalltalk to force updates to the screen. And for user input events from mouse and keyboard, TruffleSqueak instructs AWT and Swing to put all events into a queue that is frequently polled and emptied by Squeak/Smalltalk's `EventSensor`.

Performance Optimizations in TruffleSqueak

TruffleSqueak deploys a number of optimizations to improve the performance of the language and, hence, the usability and responsiveness of the programming system. Some of them are commonly known optimizations for dynamic languages, some are specific to TruffleSqueak.

Its bytecode loop is not only heavily tweaked with hints for the compiler. It also tries to keep overhead in the interpreter low, for example by avoiding boxing. Furthermore, bytecode nodes are lazily initialized for each bytecode. On the one hand, this reduces memory consumption as only AST nodes are allocated that are actually needed. At the same time, this also reduces the size of the Graal compiler's output significantly. A branch, for example, that has never been taken will also never be part of compiled code, except if it is taken at some point. While this improves compilation times and sizes and therefore warmup, this optimization can potentially lead to more deoptimizations at run-time and thus more work for the JIT compiler.

Moreover, TruffleSqueak comes with an object model that consists of more than 15 different classes for representing Smalltalk objects. Each of them is optimized to represent the corresponding type of Smalltalk object in an efficient way. `ClassObject`, for example, is used for representing Smalltalk class and additionally manages compiler assumptions about the stability of a class' method dictionary, its overall class hierarchy, and its format. `ArrayObject` makes use of storage strategies to optimize their allocations as well as the representations of their contents. Similar to other Truffle languages, TruffleSqueak also uses Java primitive types for representing boolean values, numbers, and characters as well as a singleton for its `nil` value.

The most complex optimization within the object model can be found in the three classes used for representing fixed-size, variable-sized, and weak pointers objects. For these kinds of objects, TruffleSqueak uses a custom shape-based object layout inspired by maps from Self [23] and the implementation in SOMns. Each class object for such objects manages a layout description. Such a description maintains a mapping from a slot location of the Smalltalk object to a specific field or array offset in the object model. The corresponding model classes have three inline `Object` fields, three inline `long` fields, as well as two fields for potential extension arrays, `Object[]` and `long[]`. Figure C.1 shows how we determined the number of inline `Object` fields. Slot locations start uninitialized, returning `nil` when they are read from. They can then be specialized to a Java primitive type stored as an unboxed `long` value. If specialization is not possible, slot locations are marked as generic and assigned to an `Object` field or an offset in the `Object[]` array. The goal of this optimization is to make the best use of the inline fields and extension arrays for the objects of a specific Smalltalk class. This can avoid boxing, which in turn reduces memory consumption and run-time performance. When such objects are accessed, TruffleSqueak looks up and potentially caches the layout attached to the object's class. It then knows which Java field it needs to read from or write to for the corresponding slot of the Smalltalk object. In case a value needs to be written that cannot be represented by the assigned Java

field or array offset, a respecialization is performed and the corresponding layout is evolved appropriately.

Although Truffle provides `DynamicObject`, a similar, shape-based representation for objects of guest languages [220], the implementation is designed for objects of variable size. The size of Smalltalk objects, however, is known at allocation-time. While `DynamicObject` provides performance enhancements over a simple `Object[]` representation that we could have also used for pointers objects, we found that it adds unneeded complexity and overheads. For example, it uses a map from any given key to a storage location. Smalltalk objects cannot grow or shrink, so a simple index is sufficient and more memory efficient.

Furthermore, TruffleSqueak implements numerous primitives and VM plugins that are optional, but performance-critical for some operations. It, for example, provides `FloatArrayPlugin` and `Float64ArrayPlugin` for efficient access of arrays of 32bit and 64bit floating-point numbers. Similar to BitBlt and Balloon, we manually port the C sources of the `JPEGReaderPlugin` and the `ZipPlugin` to speed up the encoding and decoding of JPEG images, ZIP files, as well as PNG images.

In Section 11.2, we evaluate TruffleSqueak's UI performance in detail. Additional performance benchmarks can be found in Appendix B.

8.2. Opening Squeak/Smalltalk to Other GraalVM Languages

The next step for turning Squeak/Smalltalk into an exploratory tool-building platform for GraalVM is to connect the language interoperability protocol. In this chapter, we explain how TruffleSqueak implements this protocol and how this allows interaction with other languages. Since language interoperability is bidirectional, we first show how TruffleSqueak provides access to other guest languages. Then, we illustrate how objects from Squeak/Smalltalk are exposed to other languages.

Providing Access to the Language Interoperability Protocol

Each GraalVM guest language provides a polyglot API that allows the execution of code written in other languages. This is usually done via a dedicated module, class, or a set of builtins. The TruffleSqueak programming environment comes with a dedicated `Polyglot` class that allows the evaluation of a string or a file written in one of GraalVM's guest languages. The following expression, for example, returns JavaScript's `Math` module: `Polyglot eval: #js string: 'Math'`. On the VM level, TruffleSqueak im-

Listing 8.1: Simplified implementation of the `ForeignObject>>doesNotUnderstand:` method.

```
doesNotUnderstand: aMessage
  | member arguments |
  member := aMessage selector asString copyUpTo: $:.
  arguments := aMessage arguments.
  (member = 'new' and: [ Interop isInstantiable: self ])
    ifTrue: [ ^ Interop instantiate: self with: arguments ].
  (Interop isMemberInvocable: self member: member)
    ifTrue: [ ^ Interop invokeMember: self member: member arguments: arguments ].
  (arguments size = 1 and: [ Interop isMemberWritable: self member: member ])
    ifTrue: [ ^ Interop writeMember: self member: member value: arguments first ].
  ^ (Interop isMemberReadable: self member: member)
      ifTrue: [ | result |
        result := Interop readMember: self member: member.
        (Interop isExecutable: result)
          ifTrue: [ Interop execute: result with: arguments ]
          ifFalse: [ result ] ]
      ifFalse: [ super doesNotUnderstand: aMessage ]
```

plements a `PolyglotPlugin` that provides corresponding primitives for the evaluation of foreign code and files.

This plugin also exposes GraalVM's message-based protocol for language interoperability directly: For each message supported by the protocol, the plugin provides an appropriate Smalltalk primitive. Within TruffleSqueak's programming environment, all of them are accessible through the `Interop` class. With `Interop hasMembers: anObject`, for example, it is possible to send a message through the interoperability protocol to check whether `anObject` has members according to the protocol.

Access to the host Java is available through another set of functions, which the `PolyglotPlugin` exposes in a similar fashion. Within the programming environment, they can be accessed through the `Java` class. For example, `Java type: 'java.lang.System'` returns the `System` class from host Java.

Furthermore, TruffleSqueak comes with a `ForeignObject` class that is used to represent objects from other guest languages and host Java. This class is registered on the VM level during startup of the environment and then used to access all foreign objects. Various core methods of Squeak/Smalltalk's `Object` class are overridden in `ForeignObject` and make use of the interoperability protocol. Instead of calling out to *primitiveAt*, `ForeignObject>>at:`, for example, sends either an interoperability message to read an array element or a member depending on whether the argument is a number or not.

An important method override is `ForeignObject>>doesNotUnderstand:`. TruffleSqueak leverages the Smalltalk `doesNotUnderstand:` mechanism [53, pp. 589–590] to forward Smalltalk messages to foreign objects: If a Smalltalk message is not understood by `ForeignObject`, its `doesNotUnderstand:` method is invoked and used to map the semantics of a Smalltalk message send to the interoperability protocol. A simplified version of this method

is depicted in Listing 8.1. First, the method converts the message's selector to an interoperability member name. For this, it simply turns the selector into a string and copies it until the first colon. Then, it loads the message's arguments into a temporary variable. Afterward, the actual dispatch begins: If the member name equals `'new'`, an *instantiate* message is sent through the interoperability protocol with the arguments provided. Otherwise, the methods sends an *invoke member* message if the object has an invocable member for the given member name. If that is not the case, the methods sends a *write member* message if and only if one argument was provided and an appropriate writable member exists in the receiver. If that is also not the case, an *is member readable* message is sent to check whether a member can be read. If that is true, a *read member* message is dispatched and if the result is executable according to the protocol, the method will also execute the result before returning it. If none of the checks succeed, the method will fall back to `Object>>doesNotUnderstand:`. This code path will open the usual debugger window in the environment to inform the user of a `MessageNotUnderstood` error.

Different parts of Squeak/Smalltalk's object protocol are also routed through the language interoperability protocol for foreign objects: The `ForeignObject>>instVarNamed:` method, for example, sends a *read member* message while `ForeignObject>>instVarNamed:put:` sends a corresponding *write member* message. Since Squeak/Smalltalk's tools for object inspection use reflection to access objects, these method overrides are crucial for tool support, as we show in Section 8.3.

Exposing Squeak/Smalltalk Objects to Other Languages

To be able to pass Squeak/Smalltalk objects to other languages supported by GraalVM, TruffleSqueak's object model needs to implement the interoperability protocol. Usually, this is done entirely on the level of the language implementation in Truffle. TruffleSqueak, however, follows a different approach: Similar to how Smalltalk messages are forwarded as interop messages, interop messages are also forwarded and dispatched on Smalltalk objects. On the language implementation level, interop messages are only translated to corresponding Smalltalk messages. Instead of doing this for every message supported by the interoperability protocol, TruffleSqueak makes use of Truffle's `ReflectionLibrary`, which allows us to redirect all messages in a single reflective *send* message.

There are two important reasons for implementing the interoperability protocol within Smalltalk: First, it allows changes of the implementation at runtime. By redefining methods used for language interoperability, it is possible

Listing 8.2: Implementation of `Interop>>isString:`.

```
isString: anObject
  <primitive: 'primitiveIsString' module: 'PolyglotPlugin'>
  ^ anObject interopIsString
```

to change how Squeak/Smalltalk objects are exposed to other languages without having to restart the entire system. And second, many higher-level types, such as iterators or dictionaries, are supported by the protocol, but unknown to the Squeak/Smalltalk VM. A `Dictionary`, for example, is built on top of `Arrays` in Squeak/Smalltalk. If a `Dictionary` runs out of free slots, it decides when and how to grow its internal `Array` as new key-value pairs are being added. To implement the hash API of the interoperability protocol, the VM implementation would not only need to know the `Dictionary` class but also about how it manages keys and values in `Arrays`. By dispatching interop messages on the object, the object can decide how to respond. Another advantage of this approach is that it allows developers to define how their domain-specific objects implement the interoperability protocol. This allows another dimension for exploration. A morph visualizing a seven-segment display, for example, could be exposed as a string when passed to another GraalVM language.

Another detail of TruffleSqueak implementation of the interoperability protocol is that it does not rely on VM support. When the VM does not provide the `PolyglotPlugin`, all corresponding primitives fail. The fallback code of these primitives can also handle interop messages. Listing 8.2, for example, shows the implementation of `Interop>>isString:`. On TruffleSqueak, the primitive would send an *is string* interop message, which either returns `true` or `false`. On an OpenSmalltalkVM, on the other hand, the primitive fails. Instead, the fallback code of the primitive calls the `#interopIsString` message on `anObject`. This makes it possible to test and explore the implementation of the interoperability protocol without even running on GraalVM.

8.3. Re-Using Exploratory Tools for GraalVM Languages

One goal of TruffleSqueak is to provide exploratory tools for GraalVM languages. Since these tools mostly operate on the reflection interface of Squeak/Smalltalk objects, which we have overridden and mapped onto the interoperability protocol in `ForeignObject`, most of their features work across languages without any further modifications.

Figure 8.2 shows that an unmodified workspace from Squeak/Smalltalk can be used to interactively evaluate code and send messages across languages.

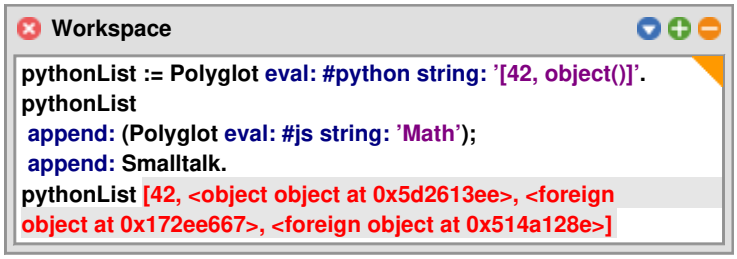
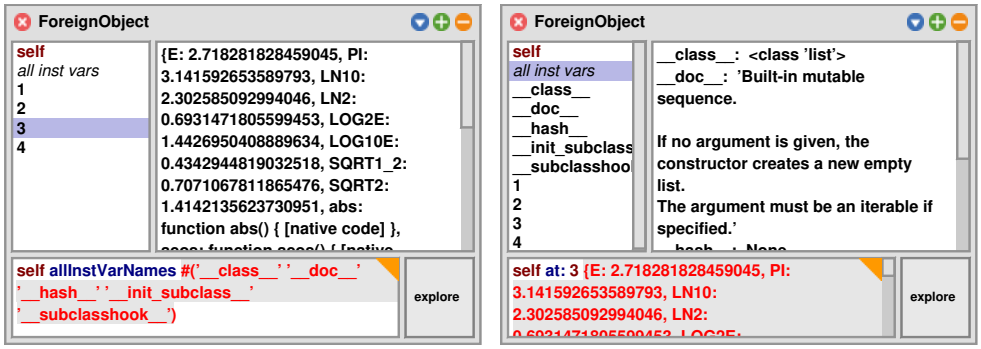


Figure 8.2.: An unmodified Squeak/Smalltalk workspace can be used to evaluate code written in different languages through the polyglot API and to send messages to foreign objects. In this example, Python code is evaluated that creates a Python `list` with two elements. Then, the JavaScript `Math` module as well as the global `Smalltalk` object representing the Smalltalk environment are appended to the list. Finally, a *printIt* reveals that the `pythonList` now contains four elements: the two Python objects and two foreign objects, the `Math` module, and the `Smalltalk` object.

Users, however, must explicitly use the polyglot API to evaluate code from other languages because the workspace itself only supports the evaluation of Smalltalk code as it directly interacts with the Squeak/Smalltalk compiler. On the other hand, it behaves in the same way and provides the same features independently from what languages are evaluated and interacted with. With a *printIt*, for example, any expression can be evaluated and the textual representation of the result is displayed.

Similarly, foreign objects can be inspected in an unmodified object inspection tool. Figure 8.3 depicts two inspectors opened on the `pythonList` object from Figure 8.2. Both inspectors automatically display the elements of the list. This is possible for two reasons: First, we have mapped the *Array* trait of GraalVM's interoperability protocol to the indexed field concept of Squeak/Smalltalk's object protocol in our `ForeignObject`. And second, GraalPython exposes this trait for Python `list` objects.

The unmodified inspector from Squeak/Smalltalk shown in Figure 8.3a, however, does not list any instance variables of `pythonList`, nor will it display any instance variables for any other `ForeignObject`. The reason for this is that it looks up the list of instance variables through the object's class, which for foreign objects is always `ForeignObject`. `ForeignObject`, however, is only used to represent foreign objects and does not have instance variables nor actual instances in the first place. While looking up instance variables in the object's class is correct for Smalltalk, other languages may allow objects to have additional, instance-specific variables. For this reason, the meta-object type from GraalVM's interoperability protocol cannot be used to look up variables and therefore cannot be directly mapped to Small-



(a) The unmodified inspector from Squeak/Smalltalk shows the elements of the `pythonList`. The third element is the `Math` module from JavaScript and the inspector reveals what kind of textual representation Graal.js provides for it through the interoperability procotol.

(b) A `PolyglotInspector`, a subclass of the inspector specifically for foreign objects, asks the object, not the class, for a list of all readable, but not invocable members and displays them as instance variables alongside the object's array elements.

Figure 8.3.: Two object inspectors opened on the `pythonList` from Figure 8.2. They allow further inspection of the `pythonList` and its elements from different languages. In the bottom part of each inspector, further interaction with the inspected object is possible.

talk's concept of classes and its meta-object protocol. To fix the inspector, we instead create a new subclass of the original inspector and change the lookup from `object class allInstVarNames` to `object allInstVarNames`. The hooks for reading and writing instance variables, on the other hand, also require the object and are therefore already implemented on the instance side and appropriately overridden in `ForeignObject`. Figure 8.3b shows a screenshot of TruffleSqueak's `PolyglotInspector`, which lists all readable, but not invocable members as instance variables of the object. By overriding `ForeignObject>>inspectorClass` so that it returns the new subclass, we can further instruct Squeak/Smalltalk to use our adapted inspector whenever a foreign object is inspected.

Although TruffleSqueak properly supports `allInstances`, `become:`, and other features that are helpful for exploratory programming for Squeak/Smalltalk, these features require far more work to be adapted to other languages. For cross-language `allInstances`, for example, it must be possible to walk all objects from all languages, not just the ones reachable from Squeak/Smalltalk. Other language implementations must provide appropriate means for this, a list of GC roots or an iterator for all objects for example. This and other limitations are left for future work and discussed in Section 11.4 in more detail.

Summary We present TruffleSqueak, an implementation of our approach for the GraalVM. It is based on Squeak/Smalltalk and consists of mainly three components:

1. an implementation of the Squeak/Smalltalk VM written as an AST interpreter in GraalVM's Truffle framework, supporting stock Squeak/Smalltalk images, all required language features, and two bytecode sets,
2. a dedicated VM plugin providing access to the language interoperability protocol of GraalVM and other components of the runtime system, and
3. Squeak/Smalltalk code that implements and integrates this protocol.

TruffleSqueak enables exploratory programming across all GraalVM languages, allowing developers to interactively evaluate code through GraalVM's polyglot API and to interact with objects from different languages. At the same time, TruffleSqueak makes it possible to use the Morphic UI framework as well as the `ToolBuilder` infrastructure from Squeak/Smalltalk to build and evolve tools for polyglot programming at run-time. TruffleSqueak is, therefore, the foundation of our fourth contribution.

9. Extending Exploratory Tools of Squeak/Smalltalk for GraalVM

With the polyglot API and our `ForeignObject` implementation, only little modification was required to make the exploratory tools of Squeak/Smalltalk work for other languages. Nonetheless, the tools only allow users to interact with other languages indirectly through Smalltalk and its polyglot API. In the following, we describe how we extend these tools so that they provide and incorporate information on language interoperability, making them aware of other languages and additional capabilities of the GraalVM.

9.1. Revealing All Interoperability Members of Objects

The inspector tool in Squeak/Smalltalk is designed to show the instance variables as well as indexed variables of an object. So far, this also applies to our subclass depicted in Figure 8.3, which shows the attributes of a Python `list` as instance variables and its items as indexed variables. For understanding an object's interface, a browser tool can be opened on the class of the inspected object from within an inspector in Squeak/Smalltalk. The browser can then be used to inspect and modify the interface of a Smalltalk object.

In Squeak/Smalltalk, however, classes are organized in a defined structure reflected in the browser tool. Lively Kernel has shown that this structure can be mapped to files, which are required to manage code in JavaScript and most other languages. Since we do not want to enforce Squeak/Smalltalk's structure for managing code across languages, we instead extend TruffleSqueak's `PolyglotInspector` so that it also lists the interfaces of objects.

The easiest way to do this would be to list all interoperability members as part of `ForeignObject>>allInstVarNames`. For a simple Python `list` object, however, this would mean that another 40 invocable members are shown. Other languages, such as Ruby and Smalltalk itself, have much larger object protocols and can expose hundreds of invocable members. Developers, on the other hand, are often interested in either the structure or the interface of an object, rarely in both at the same time. Furthermore, it is not uncommon for objects from other languages to have a large number of instance variables. In addition to that, some languages may also expose more than instance variables

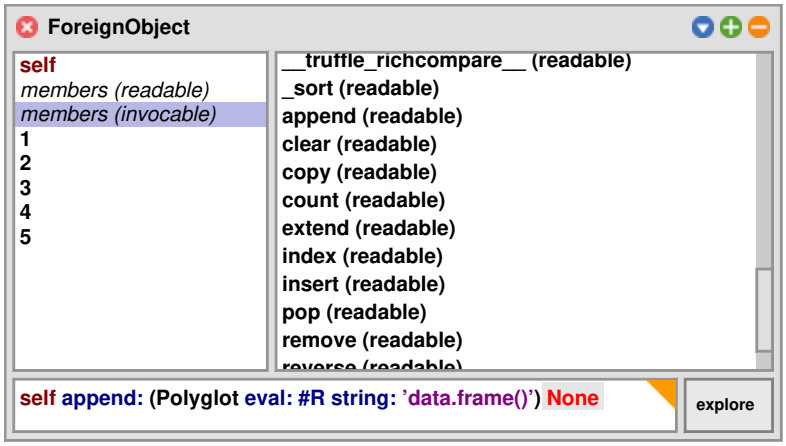


Figure 9.1.: Instead of listing readable members individually as instance variables, `PolyglotInspector` now displays two groups explicitly. The first shows a list of all readable, but not invocable members of the object. The second lists all invocable members and whether they are also readable or not. With that additional information, both the structure and the interface of an object can be explored.

as readable members. To avoid clutter and ambiguity, we thus replace the infrastructure for listing instance variables with two individual member groups, readable and invocable members. The distinction between readable and invocable members is discussed in Section 11.3. The infrastructure for indexed variables, on the other hand, can be re-used for foreign objects with large numbers of interop array elements because it supports smart truncation for large indexable objects.

Figure 9.1 shows a screenshot of TruffleSqueak's `PolyglotInspector` opened on the `pythonList` from Figure 8.2 and after implementing this additional extension: In addition to the structure of the object, the inspector now also reveals the interface of the object, which helps developers to understand how the object can be interacted with, without having to read the implementation or documentation. For example, now that it is clear that new elements can be added to the list through an `append` message, a data frame object from R can be added. The result of the *printIt* further makes clear that `append` returns `None`, unlike other languages that return the appended object or the list itself.

To provide live feedback, the Squeak/Smalltalk inspector frequently refreshes its view in case the inspected object has been changed. This mechanism is fully functional in TruffleSqueak's `PolyglotInspector` and thus available to other languages. In this example, the inspector thus lists a fifth array element almost immediately after we appended the R object.

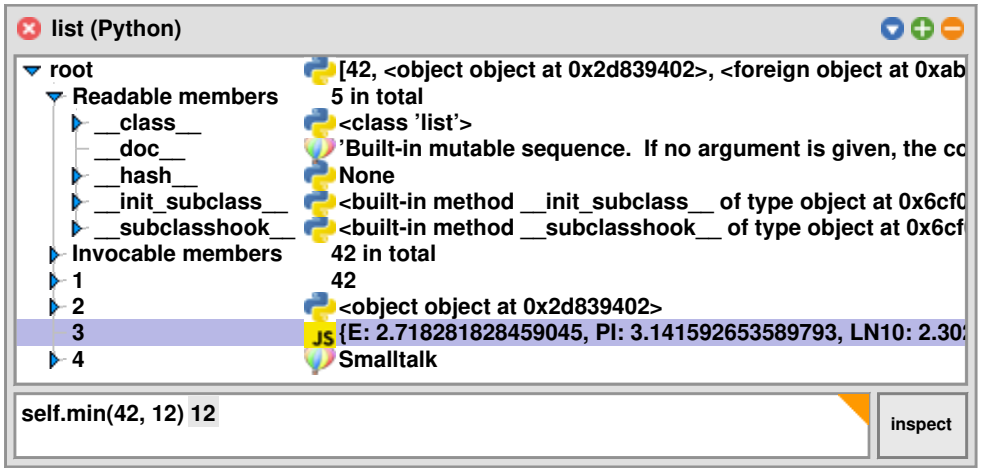


Figure 9.2.: TruffleSqueak's `PolyglotObjectExplorer` extends the object explorer of Squeak/Smalltalk and reveals information about the languages of objects. This additional information can uncover and explain unexpected characteristics of an object. In addition, the embedded workspace of the extended explorer is no longer limited to Smalltalk and allows developers to select different languages.

9.2. Providing Context About GraalVM Languages

In addition to interoperability members and array elements, we further extend the exploratory tools to provide additional context about languages. Developers, for instance, should be able to see the language of an object in the `PolyglotInspector`, for example, in the window title. Otherwise, the concrete semantics of the objects under inspection may not always be clear, which would violate the Principle of Explanation of exploratory software. At the same time, the exploratory tools of Squeak/Smalltalk are designed for and thus limited to evaluating Smalltalk code. By integrating the polyglot API and other GraalVM capabilities, we can make them aware of the polyglot environment and more convenient to use for other languages. Since we are working with a self-sustaining programming system, we are free to extend all tools and other parts of Squeak/Smalltalk through different means such as subclassing, method redefinitions, and extension methods.

A screenshot of TruffleSqueak's `PolyglotObjectExplorer`, again opened on the `pythonList` from Figure 8.2, is shown in Figure 9.2. Similar to the `PolyglotInspector`, the extended explorer tool lists readable and invocable members as well as array elements. The window label now shows the name of the inspected object's meta-object instead of its `printString`. More importantly, the language of the inspected object is explicitly mentioned in the window label as well. Items in the tree view used by the object explorer

support icons, which the `PolyglotObjectExplorer` further uses to display the language of each object. This can uncover and explain unexpected characteristics of the inspected object and therefore helps to further understand language interoperability in detail.

Although it is a Python string, the value of the `__doc__` member, for example, appears as a string from Squeak/Smalltalk. The reason for this is that Python strings are immutable and therefore, TruffleSqueak's language implementation lets them appear as Smalltalk strings automatically. This increases portability when Python strings are passed into Smalltalk code, allowing Smalltalk string methods to be invoked on the Python object.

Another interesting object is the number `42`, the first array element of the `pythonList`. According to GraalVM's interoperability protocol, it does not have a language. This is a design decision of the Truffle framework shining through our exploratory tool: Java primitive types, such as `boolean` and `double`, are often used by Truffle languages to represent booleans, numbers, characters, and other primitive guest values efficiently. While Truffle does not expose a language for these Java primitive types, it does provide an implementation of the interoperability protocol for them that is shared across all languages. Similar to Python strings, TruffleSqueak's language implementation lets these primitive values from Java appear as corresponding Smalltalk objects for portability reasons.

Moreover, the embedded workspace of the extended explorer is no longer limited to evaluating Smalltalk code. Through the context menu, it is possible to select one of the languages supported by the running GraalVM instance. Code evaluation requests are then automatically routed through the polyglot API. To provide access to the selected object of a `PolyglotObjectExplorer` or the inspected object of a `PolyglotInspector`, the tools bind "self" to the object via `TruffleLanguage.Env.parsePublic(source,names).call(args)`, Truffle's infrastructure to evaluate code with a specific local scope. This way, it is possible to interact with the `Math` module from JavaScript, for example, through Python or Smalltalk but also through JavaScript.

TruffleSqueak further supports two additional options in terms of the language selection: The default language used by our extended tools can be changed globally to any supported language in Squeak/Smalltalk's preference browser. TruffleSqueak also provides an adaptive language selection mode, which lets the embedded workspaces of the inspection tools automatically follow the language of the currently selected object. To help the user, the current language is always explicitly mentioned in the help text of such a workspace, which is shown initially and when no text is entered.

Similar to the embedded workspaces, we have also extended Squeak/Smalltalk's standard workspace tool. A screenshot of this `PolyglotWorkspace`

Figure 9.3.: TruffleSqueak's `PolyglotWorkspace` allows developers to switch between different languages. The tool's window label reveals that it is in Python mode and can therefore be used to interactively evaluate Python code without having to use the polyglot API.

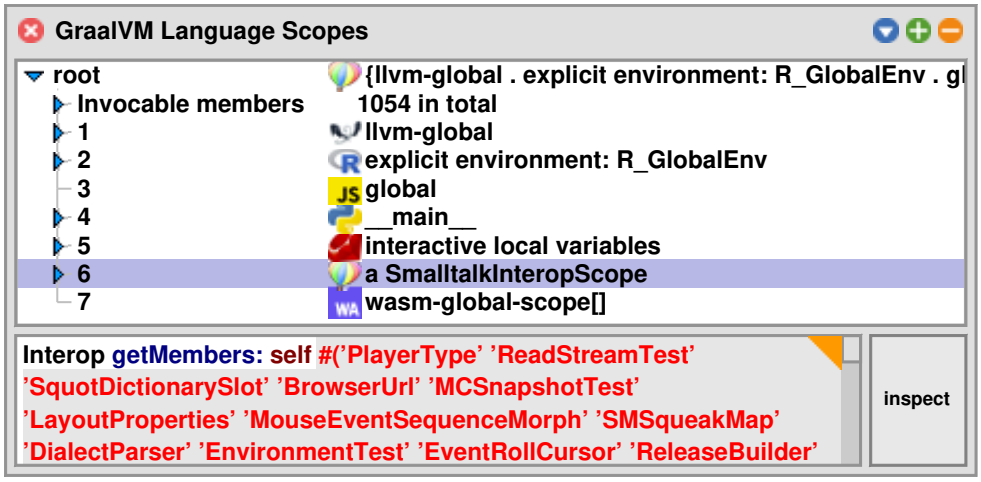


Figure 9.4.: With a simple Smalltalk expression, developers can easily explore the top scopes of all GraalVM guest languages.

is shown in Figure 9.3. Through its context menu, developers can switch between the languages supported by the running GraalVM. This hides the polyglot API from the user, making it more convenient to evaluate code from different languages. Since we only changed how the tool evaluates code, it otherwise still behaves in the same way as before.

Moreover, the ability to access the globals of a particular language provides an additional starting point for exploration and allows tools to be built that, for example, can automate sharing between languages. The closest concept to globals that Truffle provides is its support for top scopes. Each language can provide an object that describes its top scope. This API, however, is not part of the interoperability protocol. But it is accessible through Truffle's Instrument API for tools, which TruffleSqueak's language implementation also exposes to the programming system. With that, it is possible to access the top scope of a language from any tool or application and for exploration. The following

Smalltalk expression, for example, opens an explorer on the top scopes of all guest languages:

```
(Polyglot availableLanguages collect: [ :ea |
    Polyglot primitiveGetScope: ea getId asSymbol ])
        exploreWithLabel: 'GraalVM Language Scopes'.
```

A screenshot of the resulting explorer is depicted in Figure 9.4. The `SmalltalkInteropScope` provided by TruffleSqueak is implemented on the language level similar to the rest of its implementation of the interoperability protocol. It exposes all classes of the entire TruffleSqueak environment as readable members as specified by the protocol. Other language implementations such as Graal.js, FastR, and GraalWasm expose their globals through this API. Since this API is relatively new, the explorer also uncovers some inconsistencies: At the time of writing, the top scope returned by TruffleRuby is called "interactive local variables". Closer inspection reveals that this top scope, unlike the ones from other languages, has two additional parent scopes: a scope for global variables and one for Ruby's main object. Another observation is that neither the top scopes of Graal.js nor the one of GraalWasm provide any meaningful members. While both inconsistencies may be surprising for application and tool developers, they are good examples of something that needs further discussion between language developers and the maintainers of Truffle.

9.3. Incorporating Additional Features of Truffle

GraalVM's language interoperability protocol supports more types and traits than required for TruffleSqueak's exploratory programming tools, and it is still evolving. Whenever new capabilities are added to the protocol, language developers must provide appropriate implementations for their languages. To better understand how their languages as well as other languages implement certain parts of the interoperability protocol, the object inspection tools can easily be extended, similar to how we extended them with the ability to list different interoperability members.

Figure 9.5 shows a screenshot of our fully extended `PolyglotInspector`. The tool inspects a `ZeroDivisionError` from Ruby, which has different interoperability members but no array elements. According to the protocol, however, the object also provides a meta-object, a language, and implements the exception type. Information on each of these is shown in the inspector. Truffle, for example, manages additional information, such as full name, version number, and supported mime-types, for each language. This and more information are visible under "language info". The "exception info", on the other hand, shows details on how this `ZeroDivisionError` implements the

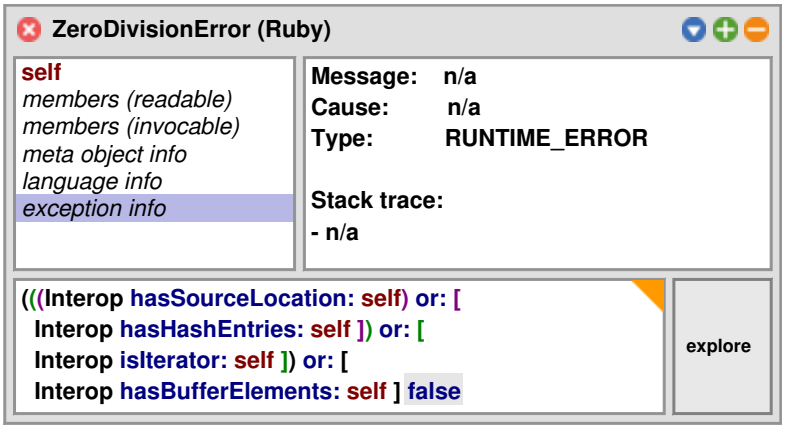
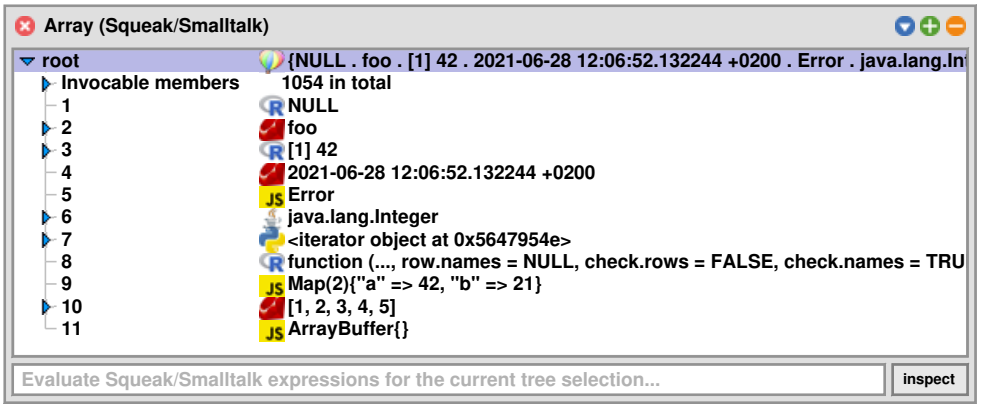


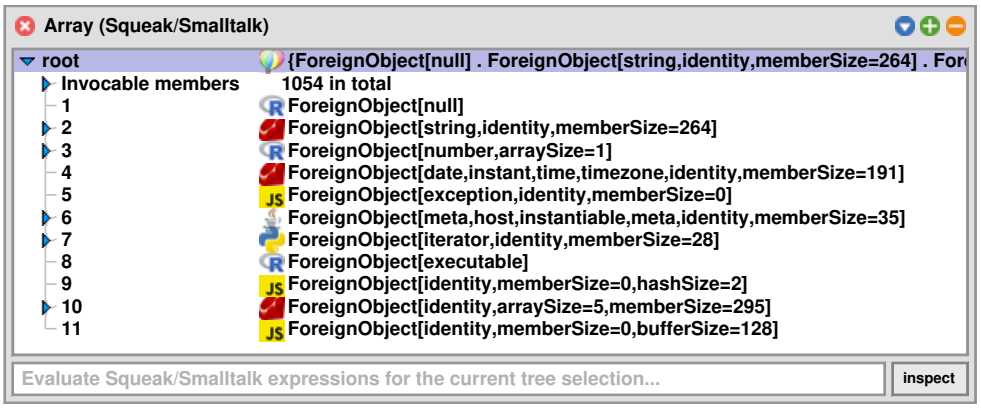
Figure 9.5.: The full `PolyglotInspector` shows additional information on how the inspected object implements interoperability traits and types. In this case, the `ZeroDivisonError` provides members, a meta-object, a language, and implements the exception type. The inspector, however, also reveals that the exception type is not fully implemented. A message, a cause, and a stack trace are not available for this object. Other interoperability traits and types, such as source location, hash entries, iterator, and buffer elements, are also not exposed by this object as the embedded workspace shows.

exception type. The only information available for this object, however, is the type of the exception. The object does not provide a message, cause, or a stack trace. For the maintainers of TruffleRuby, this type of information is helpful to understand which parts of the protocol are correctly implemented, have errors in their implementation, or are still missing. Since the `PolyglotInspector` accesses the interoperability protocol directly, new extensions only need to be implemented once and can then be used across all GraalVM languages.

Moreover, TruffleSqueak has an "emphasize language interoperability" preference that changes the textual representation of foreign objects in the entire programming system. Instead of retrieving the text through the *to display string* message of GraalVM's interoperability protocol, the text consists of the `ForeignObject` class and a list of all interoperability types and traits provided by the corresponding foreign object. This preference, therefore, gives language developers a better overview of how various objects from different languages implement the language interoperability protocol of GraalVM. Figure 9.6 shows two screenshots of a `PolyglotObjectExplorer` to demonstrate the effect of the preference: By default, the text is retrieved through the *to display string* message, which the language developers of the object's language have implemented. With the preference globally enabled, the textual representation is language-agnostic and provides an overview of what interoperability types and traits a foreign object provides.



(a) By default, textual representations for foreign objects are retrieved through the *to display string* message of GraalVM's interoperability protocol.



(b) When the "emphasize language interoperability" preference is enabled, the textual representation reveals the interoperability traits and types that foreign objects provide.

Figure 9.6.: A `PolyglotObjectExplorer` tool opened on an array of objects from different languages. The "emphasize language interoperability" preference allows developers to switch the textual representation of foreign objects globally.

Additional information on interoperability features is also helpful for the runtime developers maintaining Truffle. It helps them to better understand how different languages implement certain interoperability APIs. Since they design these APIs, they are interested in identifying potentials for inconsistencies or misuses across languages. In addition to the interoperability protocol, Truffle also provides other APIs for language and tool developers. One example is Truffle's polyglot bindings object that language developers use to implement the export-import functionalities of their language's polyglot API. By exposing it directly in TruffleSqueak, it is possible to inspect its contents at run-time. At the same time, tools like the `PolyglotWorkspace` can access and modify these bindings. And they can make them directly available to all languages during code evaluation by passing the object into the eval function of the polyglot API as a dedicated local variable. Another example is Truffle's Instrument API, which allows tool developers to instrument language interpreters, for example, to perform dynamic program analysis. This can also be accessed at run-time from within TruffleSqueak.

Summary We show how TruffleSqueak implements different extensions for the exploratory tools of Squeak/Smalltalk that make them aware of the GraalVM and its language interoperability protocol, which is an important part of our fourth contribution.

In a first step, we extend the tools for object inspection so that they not only reveal the structure of objects but also their interfaces, which helps developers to understand how to combine objects from different languages without having to read code or documentation. We also let these tools provide context about GraalVM languages, for example, by revealing the language of an object or by hiding the polyglot API from the user. Finally, we incorporate other GraalVM capabilities into TruffleSqueak such as support for meta-objects, exceptions, and other interoperability types in the object inspection tools and an "emphasize language interoperability" preference, all of which are helpful to language and runtime developers.

10. Expanding Polyglot Programming to Squeak/Smalltalk

With the extended tools for exploratory programming, it is possible to explore user applications and GraalVM components at run-time. Squeak/Smalltalk, however, provides additional building blocks for writing tools and applications that are useful in the context of GraalVM. Moreover and since Squeak/Smalltalk is a self-sustaining programming system, most of it runs on the same level as user applications on the GraalVM. As a result, polyglot programming can be applied to the entire programming system, for example, to extend it with new capabilities that are provided by libraries and frameworks written in other languages. In this chapter, we present some use cases and implementation details that give impressions of how polyglot programming can be extended to Squeak/Smalltalk: First, we discuss how TruffleSqueak allows us to apply polyglot programming to tool-building. Then, we demonstrate that TruffleSqueak allows the creation of polyglot applications at run-time. And lastly, we explain how interoperability with the host language allows further exploration of GraalVM internals.

10.1. Building Polyglot Tools for Polyglot Programming

The tools we have presented so far were entirely written and extended in Smalltalk. Since all tools run as part of TruffleSqueak's programming system and on top of GraalVM, the tools themselves can make use of polyglot programming. This allows tool developers to re-use libraries and frameworks from different languages, for example, to create visualizations, to support certain file formats, or to provide other tooling features. This way, existing Squeak/Smalltalk tools can also be turned into polyglot applications and provide realistic use cases from which we can learn more about polyglot programming.

The exploratory tools of Squeak/Smalltalk, for example, provide support for syntax highlighting, which helps developers in reading and writing code. For this, the tools use Shout, a dedicated parser for Smalltalk written in Smalltalk. To enable syntax highlighting for other languages in our extended tools, we need to either extend Shout or replace it. With Rouge, a Ruby library

Figure 10.1.: A screenshot of the `PolyglotWorkspace` from Figure 9.3. This time, syntax highlighting is provided by Rouge, a library written in Ruby.

for syntax highlighting with support for more than 100 languages, we found an appropriate replacement that TruffleSqueak allows us to re-use.

Figure 10.1 shows a screenshot of the same `PolyglotWorkspace` from Figure 9.3. This time, it uses the Rouge library from Ruby for syntax highlighting. For a given code string and a given language, the library highlights code and returns an HTML-formatted string. In Squeak/Smalltalk, the `HtmlReadWriter` allows the conversion of this string into a styled `Text` object, which is then displayed in the corresponding view. With this simple change, we have turned the extended workspace and object inspection tools into polyglot applications. Furthermore, syntax highlighting can be requested on every keystroke by an editor as the user types code. Our integration shows that it is not a problem to support such performance-critical operations through polyglot programming with GraalVM.

Squeak/Smalltalk's extensive capabilities for tool-building can further be used to extend other tools and to build entirely new tools, now with the additional ability to apply polyglot programming. Since the interoperability protocol and other language-agnostic APIs are accessible from within TruffleSqueak, new polyglot-aware tools that work across all GraalVM languages can be built for different purposes. As part of our case studies in Chapter 12, we present several tools that are implemented in a polyglot way and are designed to support developers in creating polyglot applications.

10.2. Building Polyglot Applications at Run-Time

While Squeak/Smalltalk is a good tool-building platform, it can also be used to build all kinds of other applications at run-time. The Morphic UI framework, for example, allows the construction of graphical applications such as simulations and games. Through live programming, Squeak/Smalltalk further allows developers to evolve running applications, which makes for short feedback loops compared with the conventional edit-compile-run cycle. Although this ability relies on incremental compilation and other mechanisms that are supported by Smalltalk and not necessarily by other languages, live programming can still be used when Smalltalk is involved or at least used

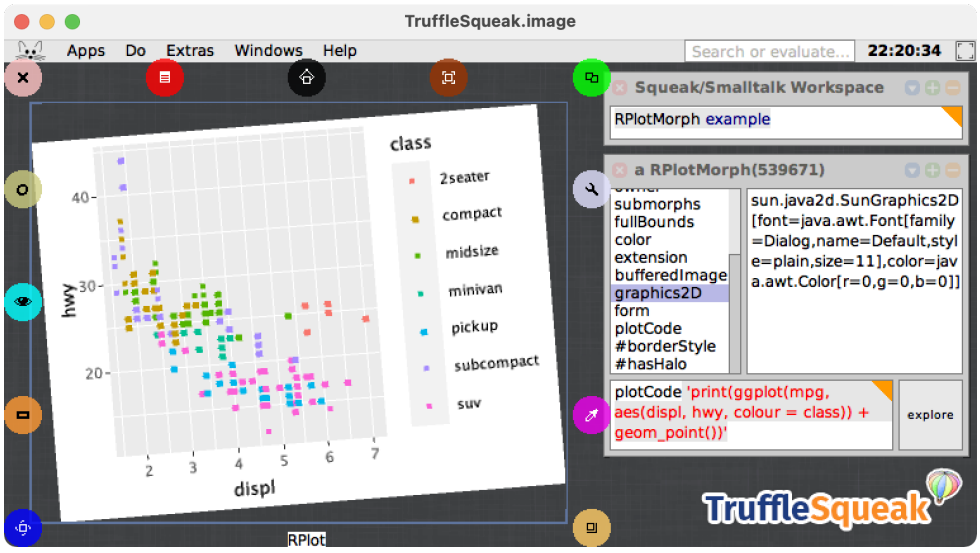


Figure 10.2.: TruffleSqueak's `RPlotMorph` is a Morphic component that renders itself through R's graphics backend. In this screenshot, an instance of this morph renders an example from the `ggplot2` package. Through Morphic's halo feature, the `RPlotMorph` can be further interacted with, for example, to resize or rotate it.

as glue between other languages. We have used this capability when we extended the `PolyglotWorkspace` with multi-language syntax highlighting, but we can also demonstrate this with an example application in Morphic:

A common functionality of UI applications, and tools for that matter, is to visualize data. For this, there are numerous visualization libraries written in different languages. Through polyglot programming, TruffleSqueak makes it possible to combine such libraries with Morphic. Figure 10.2 shows a screenshot of an `RPlotMorph` that we built in TruffleSqueak. This morph renders itself through R's graphics backend. The integration makes use of FastR's support for rendering into `Graphics2D` objects from Java's AWT framework. Instances of the `Form` class manage bitmaps that are rendered by Morphic. In TruffleSqueak's language implementation, the bitmap of a `Form` is represented by an `int[]` in Java. The missing piece to connect R's graphics backend with Morphic is a primitive that can be used to obtain a `BufferedImage` that wraps around the `int[]` of a given `Form`. By instructing the graphics backend to use the `Graphics2D` object of such a `BufferedImage`, it can render directly into `Form` objects. As the inspector to the right of the morph shows, the `RPlotMorph` holds a reference to a `BufferedImage` and its `Graphics2D` object. In addition, it also remembers the R code that is used for plotting. This is necessary to allow the morph to redraw itself, for

example when it was resized. After a resize event, the morph requests a new `BufferedImage` for the resized `Form` object, sets its `Graphics2D` as the new rendering target at the new dimension of the `Form`, and re-evaluates the plotting code. Moreover, the two Java objects that are referenced by `RPlotMorph` cannot be persisted as part of snapshots. When a Squeak/Smalltalk image is saved, these references are niled out. To remain functional after an image was loaded, an `RPlotMorph` uses the lazy-initialization pattern, which can always re-create these Java objects when they are needed for the next drawing operation.

Apart from the required primitive, we were able to build the `RPlotMorph` by evolving a running instance. With the VM introspection capabilities described in the next section, the primitive can be replaced with a Smalltalk method that performs the same operation through interoperability with the host Java language and thus does not need any modification on the level of TruffleSqueak's language implementation. The `RPlotMorph` is, therefore, an example of how TruffleSqueak enables developers to build non-trivial polyglot applications at run-time. As we show in Section 12.1 and Section 12.4, the morph is also a polyglot building block that can be re-used in other applications that want to make use of R plotting libraries.

10.3. Exploring Language Implementations and GraalVM Internals

In GraalVM, it is possible to give guest languages access to the host Java language. This is usually done through a dedicated API, in TruffleSqueak accessible through the `Java` class, that is independent of the polyglot API. This API does not allow dynamic evaluation of Java code, but it can be used, for example, to look up Java classes and to extend the Java classpath. With that, it is possible to interact with Java classes through the interoperability protocol, which allows their instantiation for example.

Figure 10.3 demonstrates the introspection of host-level objects. The language implementation of TruffleSqueak uses Java's AWT framework for drawing the programming system. The AWT `Frame` class allows enumeration of all frames through `Frame.getFrames()`. With TruffleSqueak's `Java` class, the `Frame` class can be interacted with. Since TruffleSqueak opens exactly one native window for rendering the programming environment, the `getFrames` message returns an array with exactly one element. This `Frame` instance is the very object representing TruffleSqueak's native window. This way it is possible to interact with the Java object that manages its native window from within TruffleSqueak, for example, to change the window title.

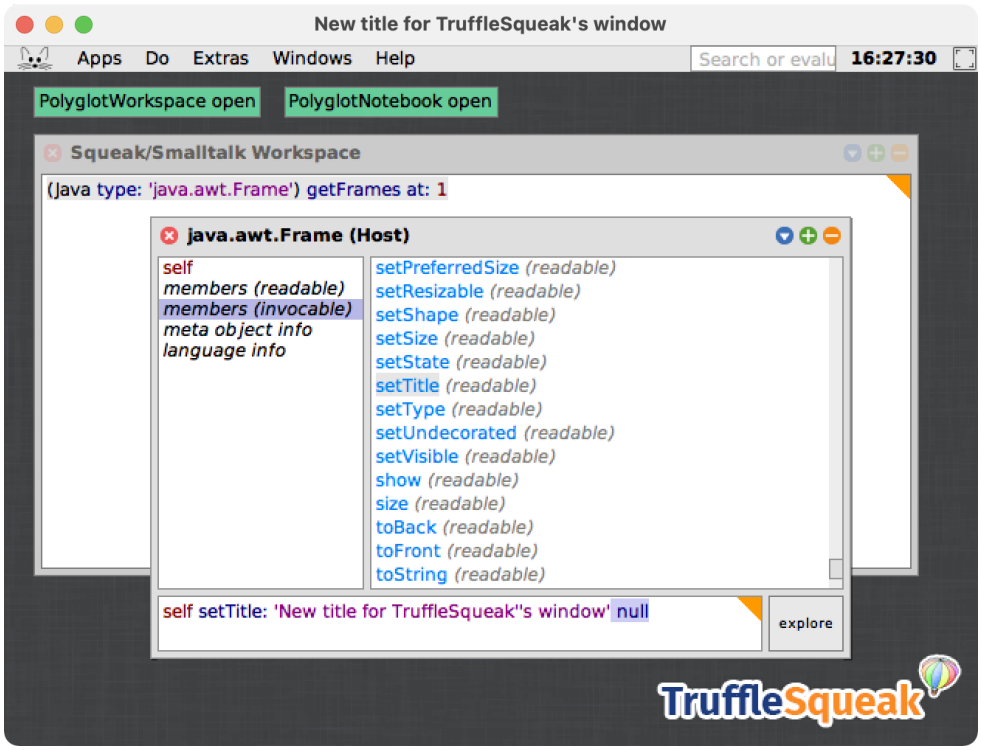


Figure 10.3.: With access to the host language, it is possible to introspect TruffleSqueak's language implementation from within itself. In this case, the first and only AWT `Frame` object is inspected. This very object represents the native window used to display TruffleSqueak's programming system and is managed in its language implementation. By invoking its `setTitle()` method, for example, we can change the window title at run-time and thus without recompiling its language implementation.

GraalVM, however, deploys a complex security model that allows fine-grained access control and sandboxing of guest languages [136]. Internally, it builds on different security mechanisms of Java, such as its module system or classloader isolation. This means that not every Java class can be looked up for security reasons, which makes it hard to explore language implementations and runtime components. Again, TruffleSqueak's language implementation can be extended with additional primitives that expose arbitrary Java classes and objects to guest languages. However, `HostObject`, Truffle's builtin facility for exposing host objects to guest languages, takes GraalVM's security model into consideration. This means that the visibility of methods, fields, and packages is strictly enforced by `HostObject`. To allow extensive exploration of host objects, TruffleSqueak circumvents these security measures: It comes with a `JavaObjectWrapper` that can be requested through an appropriate primitive

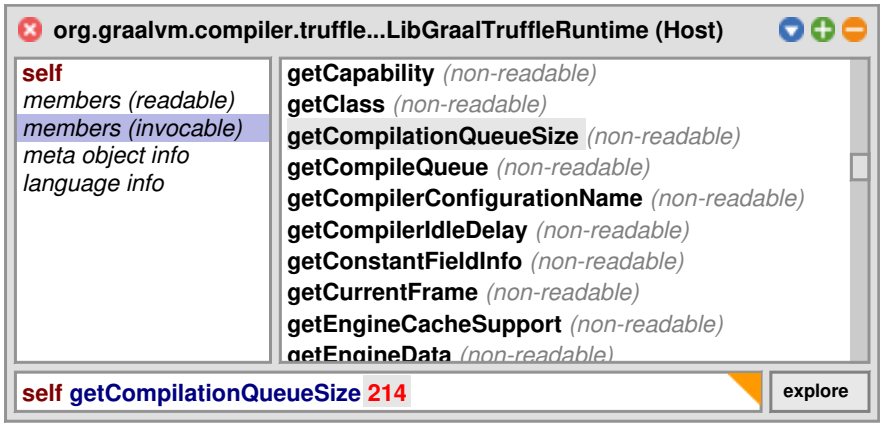


Figure 10.4.: The `TruffleRuntime` object of a GraalVM instance can be inspected at run-time. Through this object, it is possible to explore different parts of Truffle and the Graal compiler from within TruffleSqueak. In the embedded workspace, for example, the current size of Graal's compilation queue is determined.

for any Java host object. For convenience, the VM-level object for any object can further be accessed through the `vmObject` method that TruffleSqueak adds to the `Object` class. This wrapper is derived from `HostObject` but ignores visibility. Instead, it tries to open up fields and methods through Java reflection. For this to work, some Java modules and packages must be explicitly opened up through the module system, which happens as part of TruffleSqueak's launcher script. With this infrastructure, language and runtime developers can introspect and explore the internals of language implementations and the runtime, as the following example illustrates.

Figure 10.4 shows GraalVM's `TruffleRuntime` object, typically inaccessible from GraalVM languages, opened in a `PolyglotInspector`. An appropriate primitive grants access to this object through the `JavaObjectWrapper`. To make all its fields and methods visible, the package of `GraalTruffleRuntime` which implements `TruffleRuntime`, is explicitly opened up in the module system. The `TruffleRuntime` object plays a central role in GraalVM and provides access to different parts of Truffle and the Graal compiler. As the embedded workspace of the inspector shows, it is, for example, possible to determine the current size of Graal's compilation queue.

With this type of VM introspection capabilities, language and runtime developers can use TruffleSqueak to explore GraalVM internals at run-time. As we show in Section 12.4, it is possible to build tools based on these capabilities that help runtime developers to understand the behavior of the dynamic Graal compiler. To some extent, this information can also be useful

to application developers to understand how the Graal compiler optimizes their applications.

Summary We illustrate the impact of our third contribution with different examples of how polyglot programming can be applied within TruffleSqueak from the perspective of different developers:

Tool developers can re-use libraries and frameworks from other languages to provide various features in their tools. We demonstrate this with the example of syntax highlighting for multiple languages in our exploratory tools, which can be provided by a Ruby library. This simple change turns these tools into polyglot applications.

Moreover, TruffleSqueak allows application developers to build on the live programming experience of Squeak/Smalltalk and use Smalltalk to glue together different languages. This way, it is possible to build and evolve polyglot applications while they are running, which makes for short feedback loops. We illustrate this with a new UI component written in Smalltalk that uses a package from R for rendering.

Furthermore, TruffleSqueak provides extensive VM introspection capabilities based on the interoperability with GraalVM's host language. This can be used by language and runtime developers to explore the dynamic behavior of language implementations and GraalVM internals that can only be observed at run-time. We demonstrate this with two examples: We show how the native window of TruffleSqueak maintained on the language implementation level can be manipulated and how the size of Graal's compilation queue can be determined, both from within TruffleSqueak.

Part V.

Evaluation

11. TruffleSqueak: Squeak/Smalltalk on the GraalVM

In this chapter, we evaluate TruffleSqueak as a Smalltalk implementation for the GraalVM, show that it fulfills the requirements of our approach, and discuss its limitations. For this, we compare both its compatibility and performance with the OpenSmalltalkVM, a state-of-the-art Smalltalk VM and the reference virtual machine for Squeak/Smalltalk. The overall goal of this chapter is to demonstrate that platforms based on our approach can be sufficiently compatible and fast to be used for exploratory programming and tool-building. Moreover, we show that further insights about the performance characteristics of polyglot VMs such as GraalVM can be gained through self-sustaining programming systems hosted as part of guest languages.

11.1. Compatibility

Compatibility is an important aspect for users of any virtual machine. When certain language or runtime features are not correctly supported, developers may not be able to use the implementation for some use cases. In the case of TruffleSqueak, exploratory programming tools and tool-building capabilities must be fully supported for the platform to be a useful implementation of our approach.

TruffleSqueak aims to be as compatible as possible with both GraalVM and the OpenSmalltalkVM. Each release of TruffleSqueak comes with pre-built TruffleSqueak components for all operating systems, hardware architectures, and JDK versions supported by the corresponding GraalVM release. For GraalVM 21.2.0, components are available for the JDK8-based, JDK11-based, and JDK16-based GraalVM distributions for Linux (amd64 and aarch64), macOS, and Windows. In terms of the OpenSmalltalkVM, TruffleSqueak supports the latest, 64-bit Spur image format. This means that stock Squeak/Smalltalk images can be loaded and that images saved with TruffleSqueak are interchangeable with the OpenSmalltalkVM. Although not all of them are fully implemented, TruffleSqueak 21.2.0 has support for around 28 VM-level plugins and implements 741 different primitives in total. Furthermore, it also supports both bytecode sets used in Squeak/Smalltalk: the V3 bytecode set as

well as the Sista bytecode set [12]. Even though not fully tested, TruffleSqueak can also open images from Cuis, another Smalltalk dialect that runs on the OpenSmalltalkVM.

From the 4263 SUnit tests that come with Squeak/Smalltalk 6.0alpha-20288-64bit, the image on which the test image of TruffleSqueak 21.2.0 is based, 3935 tests (≈92.31 %) successfully pass on a JDK11-based GraalVM 21.2.0 and Ubuntu 20.04.2. 197 tests (≈4.62 %) are marked as passing but are not executed on our continuous integration infrastructure because they take too long to run. 40 of the remaining 131 tests are ignored for different reasons, such as incompatibilities with the headless execution mode used during testing. 31 tests are expected to fail. They are, for example, used to document specific issues within Squeak/Smalltalk. 25 tests also fail in Squeak/Smalltalk on the OpenSmalltalkVM. 18 tests (≈0.42 %) are failing only in TruffleSqueak. The remaining 17 tests are marked as flaky as they sometimes succeed or fail, for example, because they interact with weak data structures or external resources from the web.

These SUnit tests cover most of the Squeak/Smalltalk programming system including its compiler, the UI system, its standard library, its `ToolBuilder` framework, as well as almost all of its tools. Almost 97 % of all tests pass in TruffleSqueak and we found that the exploratory tools and the tool-building capabilities can be used without limitations. Therefore, we argue that TruffleSqueak's language implementation is sufficiently compatible with GraalVM and the OpenSmalltalkVM to be used for exploratory programming and tool-building.

11.2. UI Performance Evaluation

In addition to compatibility, TruffleSqueak must provide good UI performance to be used as an interactive platform for exploratory programming and tool-building. Since it is a self-sustaining programming system, the runtime performance of TruffleSqueak's language implementation has a direct impact on the performance of its programming system. In the following, we assess the UI performance of TruffleSqueak with two benchmarks. Additional benchmarks in Appendix B evaluate TruffleSqueak's language performance in more detail.

Running an IDE Workload on TruffleSqueak and GraalVM

The goal of the first benchmark is to demonstrate that the Squeak/Smalltalk programming system runs "fast enough" on TruffleSqueak to be usable. For this, we want to perform different development activities on it to create a

realistic, heterogeneous IDE workload. To quantify what "fast enough" means, we use the frame rate of the programming system as a metric. For interactive UI applications such as programming systems, a frame rate of 10 fps is often considered to be the minimum for many tasks [177, pp. 472–473, 25]. We use this value as a lower bound in the benchmark and want TruffleSqueak to provide frame rates that are well above it most of the time. Our goal in terms of UI performance was to come close to the frame rates provided by the OpenSmalltalkVM or even outperform it. For comparison, we thus want the benchmark to be reproducible on both VMs.

Setup In this benchmark, we use the `EventRecorder` tool from Squeak/Smalltalk to record mouse and keyboard events from an interactive session in which we simulate the development of a visual `CounterTool`. The `EventRecorder` allows us to prepare a Squeak/Smalltalk image that automatically re-plays these events on startup. This way, we can reproduce the same IDE workload over and over again, on TruffleSqueak as well as on the OpenSmalltalkVM. To measure the frame rate at which Squeak/Smalltalk refreshes its display buffer, we build a utility tool based on its `FrameRateMorph`. In addition to the frame rate, the tool also measures the number of compilation tasks in the Graal compilation queue through VM introspection if it runs on TruffleSqueak. This additional metric is a good indication of the activity of the dynamic Graal compiler. The frame rate, the queue size, and the current timestamp are periodically printed to `stdout` so that we can capture them over time.

Figure 11.1 shows a screenshot of the prepared benchmark image running on TruffleSqueak. In the menu bar, our utility tool displays the current frame rate and queue size in blue. On the bottom left are the `EventRecorder` and some additional buttons that we used to prepare the image. The interactive session is split into five different phases, each of which is roughly one minute long. During these phases, we perform the following development activities:

Phase 1 We start the programming system, use the workspace to open different instances of the `CounterTool`, and interact with them.

Phase 2 We inspect different aspects of one of these instances and manipulate its state.

Phase 3 We browse the implementation of the `CounterTool` and change it at run-time.

Phase 4 We repeatedly introduce errors in a method used by the `CounterTool`, trigger its execution, and recover from the errors using the debugger.

Phase 5 We combine and repeat different activities from the previous phases.

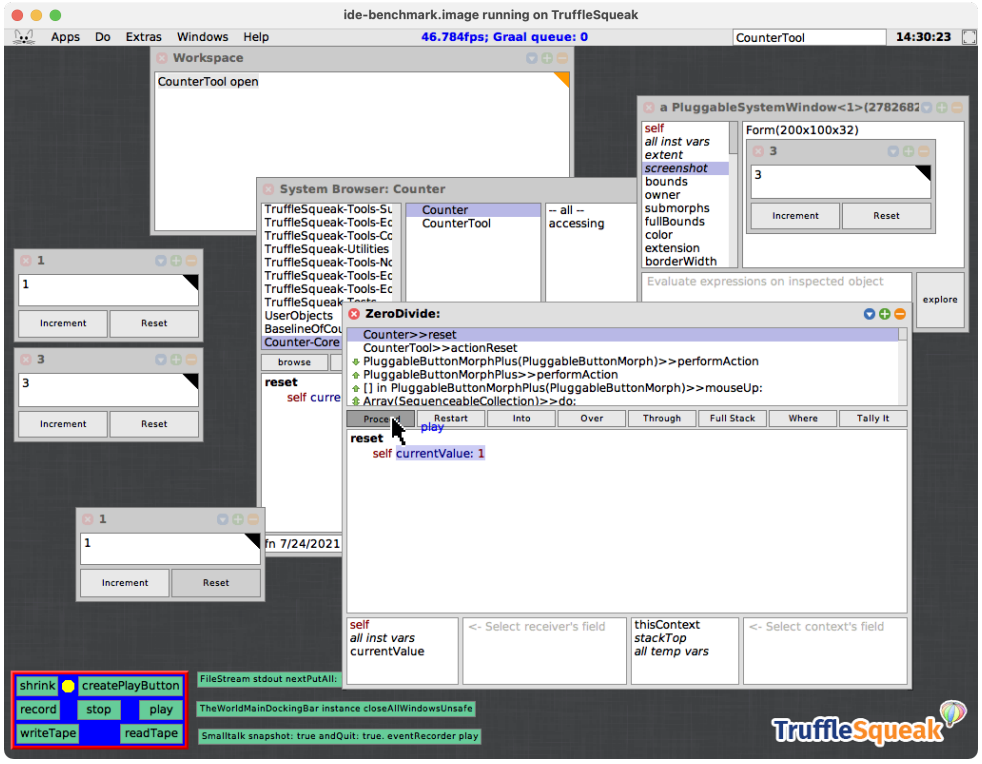


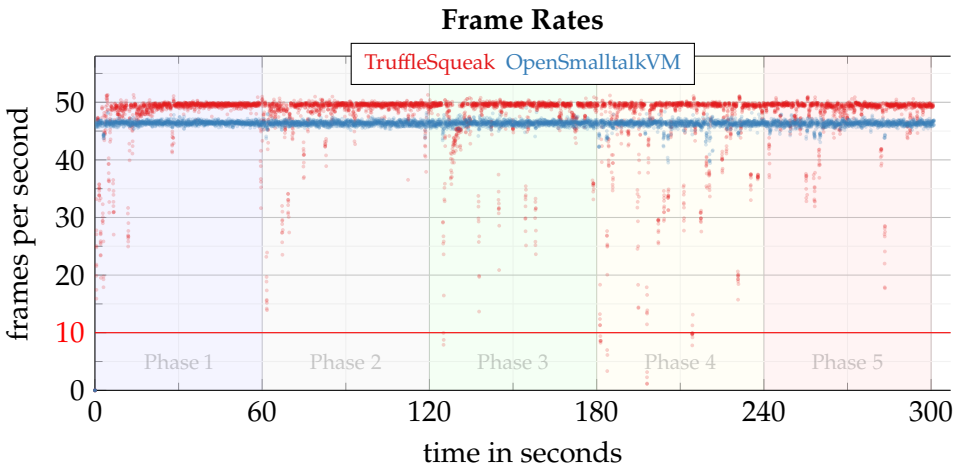
Figure 11.1.: Screenshot of the prepared IDE benchmark image running on TruffleSqueak.

The screenshot in Figure 11.1 shows instances of the `CounterTool` on the left as well as the different kinds of tools we used throughout the session. It is, therefore, taken during the last phase.

We run the IDE benchmark 10 times on each VM, TruffleSqueak 21.2.0 and the OpenSmalltalkVM 202003021730 (64-bit). In addition to the values collected with our utility tool, we use the `time` command to measure overall CPU and memory usage. To be as realistic as possible, we used an off-the-shelf developer laptop for all runs: a Dell XPS 13 7390 2-in-1 (Quad-Core Intel Core i7-1065G7 CPU @ 1.30 GHz, 31.14 GiB memory, Intel Iris Plus Graphics G7) running on elementary OS 6, a Linux distribution based on Ubuntu 20.04 LTS.

Results Figure 11.2a shows a plot of the frame rates over time comparing TruffleSqueak and the OpenSmalltalkVM, Table 11.2b lists corresponding frame rate distributions, and Figure 11.2c visualizes the size of the Graal compilation queue over time when running on TruffleSqueak.

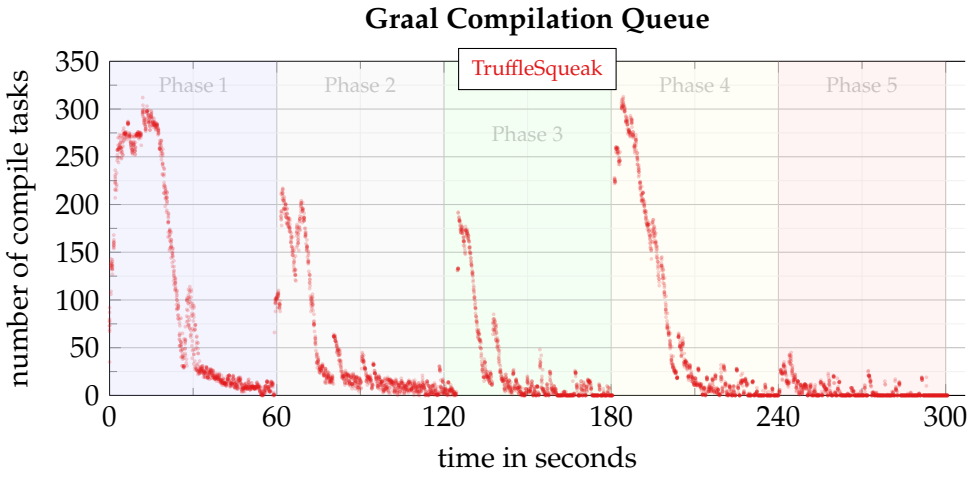
The first plot shows that TruffleSqueak reaches close to 50 fps most of the time, which is well above the minimum of 10 fps highlighted with a red line.



(a) Frame rates measured on TruffleSqueak and the OpenSmalltalkVM over time.

Interval (fps)	[0, 10[	[10, 20[	[20, 30[	[30, 40[	[40, ∞)
TruffleSqueak	0.392 %	0.750 %	1.722 %	3.853 %	93.282 %
OpenSmalltalkVM	0.000 %	0.000 %	0.000 %	0.103 %	99.897 %

(b) Distributions of the measured frame rates after the first second.



(c) Size of the Graal compilation queue over time when running on TruffleSqueak.

Figure 11.2.: Results of the IDE benchmark running on TruffleSqueak and the OpenSmalltalkVM.

Only in phases 3 and 4, the frame rate drops below 10 fps for a very short time. The OpenSmalltalkVM, on the other hand, reaches around 46 fps most of the time for the same workload. The reason the frame rate usually stays below 50 fps on both VMs is a performance option in Squeak/Smalltalk: To reduce CPU usage, Squeak/Smalltalk delays redraws of its user interface by 20 ms by default. Since its idle process consumes another millisecond, the maximum measurable value is around 47.62 fps in theory. TruffleSqueak, however, exceeds this limit continuously. We believe this is because it uses millisecond resolution for timing events while the OpenSmalltalkVM uses microsecond resolution. Consequently, timing events such as delays are less precise in TruffleSqueak. According to our measurements, this imprecision accounts for at least one millisecond, which explains why TruffleSqueak reaches close to 50 fps most of the time and slightly higher frame rates occasionally. From the user's perspective, however, this difference between TruffleSqueak and the OpenSmalltalkVM is hardly noticeable and thus neglectable.

More interesting are the raindrop-like patterns. These sudden drops in the frame rate that reoccur across runs manifest themselves as visual stutter to the user. As Table 11.2b shows, more than 6 % of all values measured on TruffleSqueak are between 10 fps and 40 fps. Nonetheless, more than 90 % of all values are above 40 fps and only less than 1 % are below the 10 fps mark. On the OpenSmalltalkVM, on the other hand, almost 99.9 % of all measured values are above 40 fps. There are mostly two reasons for TruffleSqueak's performance characteristics. One reason is that certain operations such `allInstances` or `become` are significantly slower on TruffleSqueak. In phases 3 and 5, for example, we change code, which invalidates method dictionaries and causes deoptimization in the Graal compiler. Another example is materializing stack frames, which is needed for the debuggers in phases 4 and 5. The frequent drops below 10 fps in phase 4 are also due to the debugging infrastructure becoming visible to the Graal compiler.

The second reason is warmup behavior [7], which is revealed in Figure 11.2c: At the beginning of each phase, the size of the Graal compilation queue grows substantially. As new code of the programming system is executed, more methods are identified as hot and scheduled for JIT compilation. This shows that the different development activities are rather distinct and, therefore, the IDE workload is indeed heterogeneous. Moreover, the fifth phase confirms what we expect to see: Since we did not perform any new development activities during this phase and only used tools that we used before, the warmup effect is less noticeable and substantially fewer compilation tasks are added to the queue. As a result, drops in the frame rate occur less often in the fifth phase. Nonetheless, keep in mind that the system has already been running for four minutes when this happens.

Table 11.1.: Summary of the system resource usages during the executions of the IDE benchmark shown in Figure 11.2, measured with the `time` command. For each metric, the first row shows absolute mean values and 95 % confidence intervals derived with the bootstrapping technique described in [81]. The second row shows those values relative to the OpenSmalltalkVM as a factor. The CPU load is user plus system times divided by wall-clock time. "RSS" stands for resident set size.

	TruffleSqueak	**OpenSmalltalkVM**
Wall time	306.729 s±0.068	301.409 s±0.012
	1.018±0.000	
User time	785.539 s±1.238	18.013 s±0.062
	43.609±0.172	
System time	8.294 s±0.113	5.867 s±0.059
	1.413±0.026	
CPU load	258.806 %±0.393	7.923 %±0.017
	32.666±0.089	
Max. RSS	1662.13 MB±63.766	105.83 MB±1.218
	15.701±0.670	

Moreover, TruffleSqueak requires substantially more resources for running the same workload as revealed by Table 11.1: The Graal compiler uses multiple threads by default to JIT compile methods in the background. The number of threads depends on the number of CPU cores of the machine. For running the IDE workload, TruffleSqueak uses more than 32 times more CPU resources than the OpenSmalltalkVM, which compiles methods as part of its main process instead. Similarly, GraalVM's default garbage collector allocates memory adaptively according to the running application and the amount of memory available on the machine. In this benchmark, TruffleSqueak uses more than 15 times more memory compared with the OpenSmalltalkVM: For the prepared image with a size of roughly 80 MB, TruffleSqueak allocates more than 1.66 GB of memory, while the OpenSmalltalkVM only needs around 106 MB. Furthermore, the wall-clock times show that the benchmark ran around 5 s longer on TruffleSqueak, which we believe is the result of the imprecise timing events that were mentioned before.

Overall, we believe this benchmark reflects our experience working with TruffleSqueak: Compared with the OpenSmalltalkVM, the programming system feels a bit less smooth, especially at the beginning when TruffleSqueak is warming up. Nonetheless, the system is fully functional for its intended purpose: exploratory programming and tool-building. TruffleSqueak's CPU and memory usage is significantly higher compared with the OpenSmalltalkVM, especially at the beginning, but well below the resources that modern development machines usually provide. Over time, CPU consumption usually

decreases noticeably as the system warms up and stays within reasonable bounds. This is also the reason why we neither tweaked the Graal compiler nor the GC. Both provide many options to specific limits, for example, for the number of compiler threads or the maximum heap size. The only exception is the Graal compiler mode, which TruffleSqueak sets to "latency" by default. The following benchmark analyzes the effect of this mode in more detail. The high consumption of resources and other limitations are further discussed in Section 11.4.

Evaluating Warmup and UI Performance in More Detail

In the second benchmark, we evaluate the UI performance of TruffleSqueak further and show how different modes influence warmup and peak performance. We run two experiments, one with the Graal compiler in "throughput" mode and one in "latency" mode. The former is typically used across GraalVM languages and provides higher peak performance. TruffleSqueak, however, uses the "latency" mode by default, which disables inlining and splitting, two important performance optimizations. Additionally, we also measure the performance of TruffleSqueak's AST interpreter by disabling Truffle compilation, profiling, and splitting. Note that in this configuration, the AST interpreter is still being JIT-compiled on the Java level. Furthermore, we enable the higher-performance mode in Squeak/Smalltalk, which reduces the redrawing delay from 20 ms to 1 ms. Considering the additional millisecond spent in the idle process, the maximum measurable frame rate is increased to 500 fps. At the same time, we also reduce the number of variables that could influence the performance.

Setup Similar to the previous benchmark, we prepare a Squeak/Smalltalk image and measure the frame rate of the programming system as well as the size of the Graal compilation queue. Figure 11.3 shows a screenshot of this image running at full speed on TruffleSqueak. The menu bar shows our utility tool that we also used in the previous benchmark to measure frame rates and the sizes of the Graal compilation queue over time. To avoid that any new code becomes visible to, and causes additional work in the Graal compiler over time, we do not re-play mouse and keyboard events in the programming system. Instead, we use the `BouncingAtomsMorph` from Squeak/Smalltalk, a simple gas simulation, as the benchmark application. Although this simulation is written in only around 276 source lines of code (SLOC), it exercises the Morphic UI framework in which most of the tools and other interactive components of Squeak/Smalltalk are built.

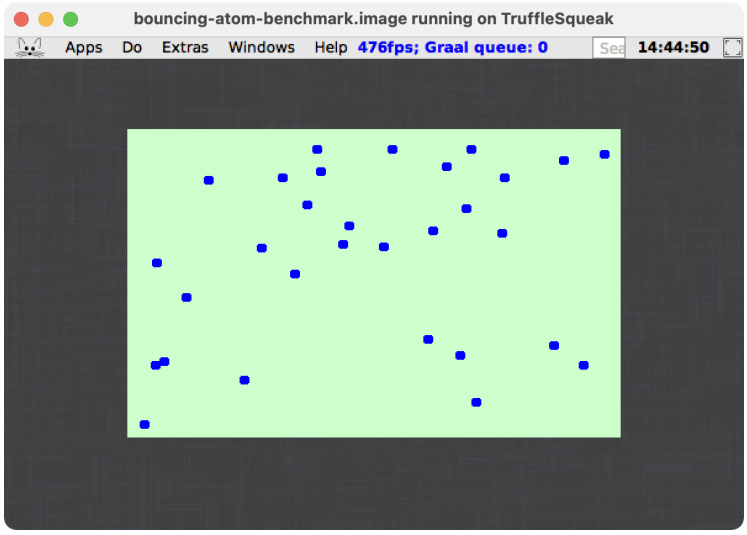


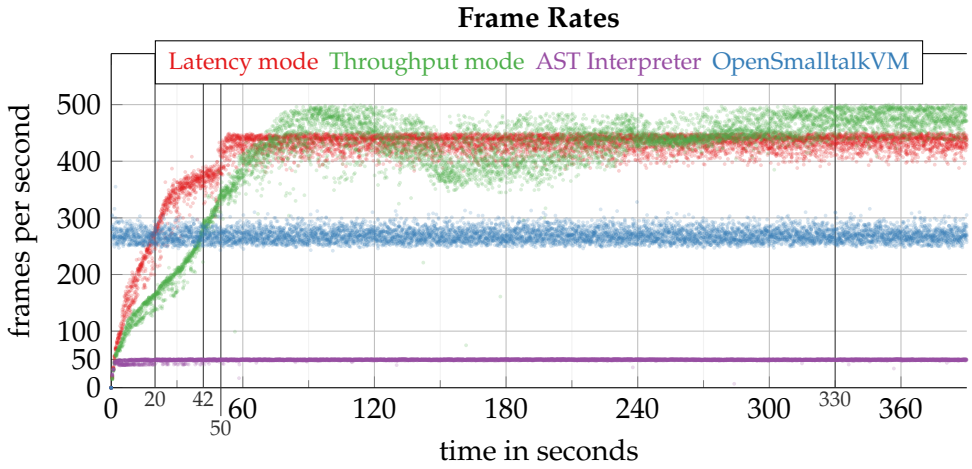
Figure 11.3.: Screenshot of the prepared UI benchmark image running at full speed on TruffleSqueak.

We run the prepared benchmark image 10 times for eight minutes on each configuration: on TruffleSqueak 21.2.0 with the Graal compiler in latency mode, TruffleSqueak with Graal in throughput mode, TruffleSqueak without Truffle compilation, profiling, and splitting, and on the OpenSmalltalkVM 202003021730 (64-bit). Instead of using a developer machine, we run all benchmarks on a dedicated benchmark server: a Dell PowerEdge 2950 (Two Quad-Core Intel Xeon E5410 CPUs @ 2.33 GHz, 32.18 GiB ECC memory) running on Debian 9. Support for hyper-threading, Intel Turbo Boost, and Intel P-States has been disabled. Since we run on a server, we use `Xvfb` to create a virtual display.

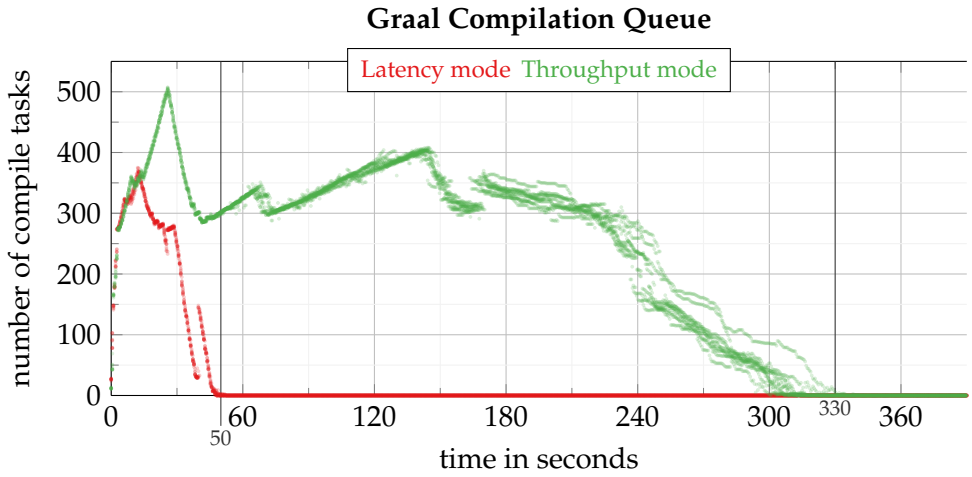
Results Figure 11.4a shows a plot of the measured frame rate values for all four configurations. In Figure 11.4b, on the other hand, only the queue sizes for the latency and throughput modes are shown. When Truffle compilation is disabled to measure the AST interpreter performance, the queue stays empty.

The first plot demonstrates that the performance characteristics of the four configurations are very different: The OpenSmalltalkVM reaches a somewhat stable frame rate range from approximately 250 fps to 290 fps almost immediately.

When the latency mode of Graal is used, TruffleSqueak reaches comparable performance after roughly 20 s and then continues to increase until it maxes out at around 400 fps to 450 fps after approximately 50 s in total. Figure 11.4b confirms that Graal has processed all compilation tasks after that time for



(a) Frame rates measured for the different configurations over time.



(b) Size of the Graal compilation queue over time when running on TruffleSqueak with the latency and throughput modes.

Figure 11.4.: Results of the UI benchmark running on four different configurations.

the latency configuration, which indicates that peak performance has been reached.

In throughput mode, TruffleSqueak needs about 42 s and thus more than twice as long to reach performance comparable to the OpenSmalltalkVM. While TruffleSqueak can reach the maximum measurable frame rate of 500 fps in that mode, it also takes much longer to reach steady performance. Although it reaches that maximum in the second minute for the first time, Figure 11.4b shows that it takes roughly five and a half minutes until Graal has processed all compilation tasks and peak performance is reached at around 445 fps to 500 fps. During that time, frame rates are high but unstable, especially in the third minute in which they drop below those of the latency configuration. Also, the queue sizes become less predictable over time across the different runs of the throughput configuration.

Furthermore, the queue sizes in Figure 11.4b show multiple spikes, for example, at around 45 s when running in latency mode. These spikes are caused by multi-tier compilation among other performance optimizations performed by the Graal compiler. To improve warmup times, Graal uses two tiers for JIT compilation by default: a fast tier that only applies some optimizations and a second tier that is slower but optimizes methods more aggressively.

TruffleSqueak's AST interpreter is with a peak frame rate of roughly 50 fps the slowest of the four configurations. On the other hand, the frame rate stays very close to 50 fps and is therefore much more consistent compared with the others. Even though profiling and splitting are disabled in this configuration, some minor warmup behavior is noticeable in the first minute. We believe this is due to the initialization of different caches, several AST respecializations in TruffleSqueak, and possibly also due to JIT compilation on the Java level.

Table 11.2 gives an overview of the resource usage of each configuration. As in the previous benchmark, all TruffleSqueak configurations use significantly more CPU and memory resources compared with the OpenSmalltalkVM. The overall CPU load when running the benchmark on the OpenSmalltalkVM is around 25 %. The load caused by TruffleSqueak in latency mode is more than five times higher, and almost 11 times higher in throughput mode. Running TruffleSqueak without Truffle compilation causes the lowest load and requires the least memory. With more than 1.9 GB, however, it still needs more than 25 times more memory than the OpenSmalltalkVM.

Overall, this benchmark illustrates how the Graal compiler influences the overall UI performance of a self-sustaining programming system, or in a more general sense, the performance of UI applications written in GraalVM guest languages. In UI applications, fast warmup and predictable performance are more important than peak performance. This is also why we decided

Table 11.2.: Summary of the system resource usages during the executions of the UI benchmark shown in Figure 11.4, measured with the `time` command. For each metric, the first row shows absolute mean values and 95 % confidence intervals derived with the bootstrapping technique described in [81]. The second row shows those values relative to the OpenSmalltalkVM as a factor. The CPU load is user plus system times divided by wall-clock time. "RSS" stands for resident set size, "OSVM" for OpenSmalltalkVM.

	TruffleSqueak			**OSVM**
	Latency Mode	*Throughput Mode*	*AST Interpreter*	
Wall time	480.209 s±0.021	480.209 s±0.015	480.168 s±0.017	480.000 s±0.000
	1.000±0.000	1.000±0.000	1.000±0.000	
User time	590.452 s±1.252	1253.102 s±14.825	496.853 s±0.783	95.683 s±1.084
	6.177±0.081	13.104±0.224	5.196±0.069	
System time	67.046 s±1.014	51.281 s±2.803	9.907 s±0.738	24.685 s±0.335
	2.716±0.059	2.080±0.112	0.401±0.031	
CPU load	136.919 %±0.183	271.628 %±3.126	105.538 %±0.075	25.077 %±0.170
	5.461±0.046	10.828±0.158	4.210±0.031	
Max. RSS	2191.46 MB±14.593	2365.04 MB±18.420	1902.18 MB±24.445	75.14 MB±0.031
	29.164±0.201	31.475±0.250	25.315±0.355	

to use Graal's latency mode by default in TruffleSqueak. Although Graal's throughput mode can provide even better peak performance, it requires considerably more work and time to reach it.

11.3. Requirement Evaluation

With the assessment of TruffleSqueak's compatibility and performance, we have demonstrated that the exploratory programming tools and the tool-building capabilities can be used in their original form on top of the GraalVM. In this section, we show that TruffleSqueak also fulfills the requirements of our approach, allowing it to be used across the languages supported by GraalVM.

Table 11.3 provides a side-by-side overview and comparison of the API requirements described in our approach, the corresponding API provided by GraalVM, and the API in Squeak/Smalltalk. The table is divided into two parts: The upper part shows the required API hooks for exploratory programming tools described in Section 5.4. In the lower part of the table are the hooks required for our extensions proposed in Section 6.2.

Furthermore, we have used mathematical operators to compare the APIs from our approach with GraalVM's language interoperability API in the column "C1", and the GraalVM API with the API from Squeak/Smalltalk in the column "C2":

Table 11.3.: Side-by-side overview and comparison of the APIs of our approach, GraalVM, and Squeak/Smalltalk. Two API hooks are either equal (=) or similar (≈) in functionality, incompatible (≠), or one is a superset of another (⊃).

Exploratory Programming API		**C1**	**GraalVM Language Interoperability API**	**C2**	**Squeak/Smalltalk API**
`evaluate(language,code)`		=	`TruffleLanguage.Env.parsePublic(source).call()`	⊃	`Compiler>>evaluate:`
`printString(obj)`		=	`InteropLibrary.toDisplayString(obj)`	=	`Object>>printOn:`
`areIdentical(obj1,obj2)`		=	`InteropLibrary.isIdentical(obj1,obj2,interopLib)`	=	`Object>>==`
`getMetaObject(obj)`		≠[a]	`InteropLibrary.getMetaObject(obj)`	≠[b]	`Object>>class`
`isInstance(obj,meta)`		=	`InteropLibrary.isMetaInstance(meta,obj)`	=	`Object>>isKindOf:`
`createInstance(meta)`		=	`InteropLibrary.instantiate(meta)`	≠[b,c]	`Behavior>>new`
`listProperties(obj)`	*named*	≈	`InteropLibrary.getMembers(obj)`	≠[b]	`Behavior>>allInstVarNames`
	indexed	=	`InteropLibrary.getArraySize(obj)`	=	`Object>>basicSize`
`readProperty(obj)`	*named*	≈	`InteropLibrary.readMember(obj,member)`	≈[c]	`Object>>instVarNamed:`
	indexed	=	`InteropLibrary.readArrayElement(obj,index)`	=	`Object>>basicAt:`
`writeProperty(obj)`	*named*	≈	`InteropLibrary.writeMember(obj,member,value)`	≈[c]	`Object>>instVarNamed:put:`
	indexed	=	`InteropLibrary.writeArrayElement(obj,index,value)`	=	`Object>>basicAt:put:`
`listMessages(obj)`		≈	`InteropLibrary.getMembers(obj)`	≠[b]	`Behavior>>selectors`
`send(obj,msg,args)`		≈	`InteropLibrary.invokeMember(obj,member,args)` `InteropLibrary.execute(InteropLibrary.readMember(obj,member),args)`	≈	`(Object>>perform:withArguments:)` `Object>>doesNotUnderstand:`
API of Proposed Extensions					
`getLanguage(obj)`		=	`InteropLibrary.getLanguage(obj)`		*no equivalent*
`listLanguages()`		=	`TruffleLanguage.Env.getPublicLanguages(obj,args)`		*no equivalent*
`evaluate(language,code,locals)`		=	`TruffleLanguage.Env.parsePublic(source,names).call(args)`	⊃	`Compiler>>evaluate:in:to:`
`listGlobals(language)`		≈	`TruffleInstrument.Env.getScope(languageInfo)`	⊃	`Environment>>keys`
`getGlobal(language,name)`		≈	`InteropLibrary.readMember(scope,member)`	⊃	`Environment>>at:`
`setGlobal(language,name,value)`		≈	`InteropLibrary.writeMember(scope,member,value)`	⊃	`Environment>>at:put`

[a] Incompatible because metaobjects are optional according to GraalVM's language interoperability API.
[b] Incompatible as a result of [a].
[c] Additionally handled in `ForeignObject>>doesNotUnderstand:` (see Listing 8.1).

- The "=" operator means that two API hooks are equal in functionality.
- The "≈" operator indicates that two hooks are similar so that hooks from one API can be mapped to appropriate hooks from the other.
- The "≠" operator implies incompatibilities between hooks, which require tools to be adapted.
- The "⊃" operator denotes that one hook is a superset of another, which also requires adaptations.

As shown in the table, GraalVM's language interoperability protocol sufficiently covers the API requirements for both, exploratory programming as well as our proposed extensions. The only exception is `getMetaObject` because, in GraalVM, the hook is allowed to throw an error if an object has no known metaobject. Similar but not equal in functionality are GraalVM's API hooks to access the structures and interfaces of objects, to send messages, and to access language globals. GraalVM's interoperability protocol only supports *named* and *indexed* object properties. Other types of properties are not supported. Indexed properties map well to the array trait of the interoperability protocol, which in turn maps well to indexable variables in Squeak/Smalltalk.

Truffle's member trait, on the other hand, distinguishes between several different types of members, such as *readable* and *invocable* members among others. Because of that, it can be ambiguous whether a member is a named property (readable) or a message (invocable) because they can be both, readable and invocable, at the same time and return two different things (e.g., the value of a property and a bound method respectively). In TruffleSqueak, we decided to map readable but not invocable members to the concept of named variables in Squeak/Smalltalk. This, however, requires an additional check of all members to split the list into two, readable but not invocable members (named variables) and invocable and possibly readable members (messages). The member trait is also ambiguous when it comes to sending messages: Members may not be invocable, but readable members may return objects that are *executable* according to the interoperability protocol. While this is seldomly the case, language developers are allowed to expose the objects of their languages in that way.

As discussed in Section 9.2, GraalVM allows tools to access the top scope of each language. The idea behind top scopes is similar to our concept of language globals. GraalVM languages are, however, allowed to expose more than their globals through this API. TruffleRuby, for example, returns a top scope object that contains interactive local variables, global variables, and the members of Ruby's main object. Since top scopes must use the member trait, they can be accessed in the same way as other foreign objects.

Moreover, GraalVM's protocol for language interoperability is mostly compatible with the API from Squeak/Smalltalk. In Section 8.3, we demonstrated

that most of the exploratory tools of Squeak/Smalltalk can be used without any modification. Some hooks from Squeak/Smalltalk, however, are too specific to Smalltalk to be re-used and thus provide only a subset of the required functionality. The `Compiler` interface, for example, only allows the evaluation of Squeak/Smalltalk code. As described in Section 9.2 for example, tools can be adapted to use the API hooks from GraalVM for polyglot access. Similarly, the two hooks to access languages have no equivalent in Squeak/Smalltalk, which requires tools to be adapted to take advantage of them.

Furthermore, there are four incompatibilities due to a mismatch between GraalVM's concept of metaobjects and the Smalltalk object model [53, pp. 269–272]: `Object>>class` cannot be mapped to GraalVM's `getMetaObject` because Smalltalk objects must always have a metaobject, while metaobjects are optional in GraalVM. We decided against modifying `ForeignObject>>class`. Instead, tools must request metaobjects through the interoperability protocol explicitly and handle the case when no metaobject is available. In addition, instance variables of an object and the selectors that it understands are looked up in the object's class in Squeak/Smalltalk, not in the object itself. To resolve these incompatibilities, however, only little work was required to adjust the affected exploratory tools. In Section 8.3, we illustrate how this was done with appropriate subclasses.

For sending messages to foreign objects, TruffleSqueak does not override `Object>>perform:withArguments:`, which is comparable to the `send` hook for exploratory programming. This would require users to always use the `perform:withArguments:` meta-programming facility explicitly. Instead and as discussed in Section 8.2, TruffleSqueak leverages the `doesNotUnderstand:` mechanism from Squeak/Smalltalk, which allows users to directly dispatch messages to foreign objects. This mechanism is further used to provide shortcuts for instantiating metaobjects and for accessing instance variables of objects, as *c* indicates in Table 11.3. A simplified implementation of this `ForeignObject>>doesNotUnderstand:` method can be found in Listing 8.1.

Although supported by Squeak/Smalltalk, GraalVM's language interoperability protocol does not support any of the optional exploratory programming features described in Section 5.4. This and other limitations of TruffleSqueak are discussed in more detail in the following section.

In this and the previous sections, we have shown, at least in theory, that TruffleSqueak is sufficiently compatible with GraalVM and Squeak/Smalltalk, fast enough to host its programming system, and that it fulfills the requirements of our approach. With the case studies that we present in Chapter 12, we further provide practical evidence that TruffleSqueak can be used as an exploratory tool-building platform for the GraalVM.

11.4. Limitations

Although TruffleSqueak is a functional implementation of our approach, it comes with different limitations concerning the approach, Truffle and GraalVM, and Squeak/Smalltalk. These are now discussed in more detail.

Optional Exploratory Programming Features Some programming languages support mechanisms that allow objects to have innumerable properties and understand an arbitrary number of messages. In Python, for example, property access can be customized by overriding the `__getattr__()` and `__getattribute__()` methods. These methods can also return Python `callables`, which can be used to allow Python objects to understand an arbitrary number of messages. The same is possible with `doesNotUnderstand:` in Smalltalk and `method_missing()` in Ruby. Truffle's member trait, however, can only be used to expose known properties and messages. Therefore, language-agnostic checks for innumerable properties and messages cannot be supported in TruffleSqueak. Users can, however, fall back to checking language-specific mechanisms in some cases.

Furthermore, GraalVM's language interoperability protocol does not support cloning objects across languages, which would make it easy to create copies for exploration. In some cases, it is possible to create copies manually by creating a new instance of an object's metaobject and copying over values for all readable but not invocable members. While this is not always possible, for example, if languages choose to hide certain properties, users can fall back to language-specific mechanisms for copying objects.

TruffleSqueak properly supports the `allInstances` and `become` mechanisms for Smalltalk, which correspond to the optional `findAllInstances` and `swap` exploratory tools described in our approach. For that, it manually walks all Smalltalk objects reachable from Squeak/Smalltalk's special objects array, as described in Section 8.1. It would be possible to extend its object walking algorithm so that it traces across other GraalVM languages based on the language interoperability protocol. This, however, would mean that only objects that are reachable from Smalltalk can be found. To find all objects, language implementations need to provide similar functionality or at least a hook that provides their GC roots that can be used as additional starting points for a language-agnostic object walking algorithm. Nonetheless, these exploratory tools are typically implemented as part of the garbage collector of a VM, but neither Java nor Truffle provide appropriate APIs to interact with the GCs from the JVM.

Further Differences Between Truffle and Squeak/Smalltalk Smalltalk is an interesting edge case for Truffle in many regards. A simple example is the fact that it provides an interactive user interface. Although not supported in any way by Truffle, it is possible to implement the needed infrastructure based on UI frameworks from Java. More interesting differences are due to some assumptions and design decisions made in Truffle: Truffle languages must, for example, provide a hook for parsing that turns a `ParsingRequest` for the language into a corresponding `CallTarget`. Here, Truffle assumes that the parser for the language is part of the language implementation, which is true in many cases. In Squeak/Smalltalk, however, the parser is part of the self-sustaining programming system and therefore implemented in Smalltalk. TruffleSqueak could implement its own parser for Smalltalk code, but that would break the idea of a self-sustaining programming system. Instead, it calls out to Squeak/Smalltalk code to use its parser to generate a `CompiledMethod` object, for which it then creates a call target in response to parsing requests. Implementing Truffle APIs with appropriate functionalities from Squeak/Smalltalk is, however, not always possible. An example of this is Truffle's API for source locations. This API is required for instrumentation and allows tools, such as debuggers and profilers, to look up the sources for a given Truffle AST node. In Squeak/Smalltalk, the `#getSource` message can be sent to compiled methods to retrieve their source code, independently of whether it is stored in the changes file or needs to be reconstructed with its decompiler. This infrastructure, however, cannot be used to implement the source location API of Truffle because it would cause the execution of more guest code in Truffle. This in turn can potentially trigger new additional calls to the source location API, which ultimately result in infinite recursions during instrumentation. Due to time constraints, TruffleSqueak works around this problem by using its bytecode decoders to generate a formatted string of bytecodes for compiled methods, similar to the string used to display bytecodes in the browser tool from Squeak/Smalltalk.

Another set of problems arise from the choice of Java as the host language: For example, Java neither supports resumable exceptions nor continuations. Since both need to be supported in TruffleSqueak, alternative implementation strategies are needed. They, however, require additional work, make the language implementation more complex, and introduce performance overheads. More importantly, Truffle assumes that exceptions from guest languages generally unwind the stack. This means that the exception model from Squeak/Smalltalk breaks the moment another guest language is involved because non-Smalltalk stack frames cannot be resumed after the stack was unwound on the Truffle level. This is one reason why TruffleSqueak's debugger cannot be used to debug foreign code. Another reason is a mismatch in the execu-

tion model of Smalltalk and Java: As a self-sustaining programming system, Squeak/Smalltalk uses a language-level scheduler that controls the execution of language-level processes. Since these processes run as part of green threads, rather than in separate OS-level processes, TruffleSqueak is unable to control the execution when another guest language is running. Starting a foreign, long-running application, such as a web server, through its polyglot API in the main thread will freeze the programming system until the application terminates. It is, however, possible to run code from other languages in separate threads. Since debugging was not a main concern of this work, we did not further work on allowing TruffleSqueak's debugger to be used across other languages due to time constraints. However, we believe this could be done by running TruffleSqueak in a dedicated Java thread, allowing its UI process to run and debug code on the main thread. Similarly, the idea of Multiprocessor Smalltalk [145] could be applied to TruffleSqueak so that Smalltalk processes run, for example, in separate Java threads to enable debugging. In Section 13.1, we present another implementation of our approach that has support for cross-language debugging because we were in control of the execution model.

Another limitation is related to the snapshotting mechanism from Squeak/Smalltalk. Since most languages have no support for persistent object memory, it is hard to persist non-Smalltalk objects as part of the image snapshots. In TruffleSqueak, references to foreign objects are simply niled out. When an image is saved in which, for example, an inspector is opened on a Python object, the inspector will show the `nil` object the next time the image is loaded. While it would be possible to extend the Squeak/Smalltalk image format with support for other languages, re-instantiating objects from other Truffle language implementation requires additional work. More importantly, however, some types of objects, such as objects representing external resources such as sockets or file handles, cannot simply be re-instantiated. Extending languages with proper support for persistent object memory requires further research, which is out of the scope of this work.

Incompatibilities and Performance Concerns As mentioned in Section 11.1, TruffleSqueak is not fully compatible with the OpenSmalltalkVM. Some language features, such as object pinning, immutability, or image segments, are not fully supported due to time constraints. The same is true for some VM-level plugins, such as the `FFIPlugin` and the `OSProcess`. At the same time, these features and plugins are neither required to use TruffleSqueak for exploration nor tool-building. Furthermore, TruffleSqueak cannot fully implement the GC interface of the OpenSmalltalkVM because it must use the GCs from the JVM. Another cause for incompatibilities is TruffleSqueak's interrupt handler: The execution model of Squeak/Smalltalk requires fre-

quent checks for interrupts, for example, for scheduling and timing events. The OpenSmalltalkVM performs these checks frequently after activations of methods and block closures as well as on backward jumps. TruffleSqueak, on the other hand, checks for interrupts only before activations of methods with more than 32 bytecodes. A process switch triggered by the interrupt handler forces the entire stack to be materialized to the heap, which is an expensive operation in Truffle, especially when processes are switched frequently. Moreover, the materialization of stack frames to Smalltalk `Context` objects can also trigger deoptimizations in the compiler in some cases. Therefore, the reduced number of checks for interrupts is a performance tradeoff in TruffleSqueak. On the other hand, this means that timing events, as briefly discussed in Section 11.2, are less precise. Since TruffleSqueak does not perform such checks after backward jumps also implies that it is impossible to interrupt tight loops, in which only primitive methods are sent. This shortcoming in turn is the reason why some of the SUnit tests from Squeak/Smalltalk do not terminate on TruffleSqueak because they test the interruption of empty or tight loops.

In Section 11.2, we also found that TruffleSqueak requires significantly more CPU and memory resources than the OpenSmalltalkVM for running the same image. This is true in general for mostly three reasons:

1. By default, GraalVM uses multiple threads for JIT compilation. The OpenSmalltalkVM JIT compiles code from Squeak/Smalltalk as part of the main process and requires less CPU cycle, possibly because it optimizes code less aggressively compared with Graal.
2. In TruffleSqueak, each Smalltalk object must be represented by a corresponding Java object on the Truffle level. Java objects are, however, roughly three to four times larger than those of the image format used by the OpenSmalltalkVM.
3. The amount of memory allocated by the GCs from the JVM often depends on the available memory of the machine and is often much higher by default compared to the GC from the OpenSmalltalkVM.

This also explains why the memory consumption we reported in Section 11.2 is much higher than a factor in the range of three to four. Since GraalVM is based on the JVM, its installation sizes are also considerably larger compared with the OpenSmalltalkVM: A stock, JDK11-based GraalVM CE 21.2.0 installation for macOS, for example, is more than 880 MB in size with Graal.js being the only pre-installed language, while the file size of the OpenSmalltalkVM used for the benchmark in Section 11.2 is only 3.6 MB.

In addition, TruffleSqueak uses the latency mode of the Graal compiler by default. While this is the only compiler configuration changed in TruffleSqueak, the latency mode has a significant impact on both the overall run-time perfor-

mance and warmup: The mode disables inlining and splitting in the compiler, two important compiler optimizations that enable additional optimizations. Due two this, peak performance is reduced but so are the sizes of compiled code and the work for the compiler. At the same time, the decreased amount of work leads to noticeable better warmup behavior, which is preferred in UI applications such as TruffleSqueak. In Section 11.2 and Appendix B, we evaluate the effect of the latency mode in more detail.

Other Truffle-based language implementations face similar challenges and limitations. The GraalVM team is aware of them and is working on different ideas to resolve them in the future. One approach to improve warmup, for example, is to snapshot compiled code similar to Sista [13]. This way, the compiler does not have to optimize the parts of a language ecosystem that never or rarely change, such as a standard library, from scratch on every startup. Moreover, memory footprint of GraalVM languages can also be reduced through AOT compilation with GraalVM Native Image. Preliminary experiments suggest that this is also true for TruffleSqueak. Another improvement is the support for language safepoints [140] that was added to Truffle in GraalVM 21.1. These safepoints are designed to interrupt the execution of guest languages and could help to improve TruffleSqueak's interrupt handler. Similarly, fibers and continuations in Java, as developed as part of Project Loom [134], could further be used to make switching between Smalltalk processes more efficient in TruffleSqueak. Overall, we believe that the technical limitations of TruffleSqueak are interesting challenges for future work.

Summary TruffleSqueak is both sufficiently fast and compatible with Squeak/Smalltalk and GraalVM to be used as an exploratory tool-building platform. Two benchmarks demonstrate that its UI performance is comparable to the OpenSmalltalkVM.

Although TruffleSqueak can even outperform the OpenSmalltalkVM in some cases at the cost of higher CPU and memory consumption, the two benchmarks also illustrate that UI applications are particularly sensitive to slow warmup behavior and performance cliffs that can be caused by the Graal compiler. While this may seem plausible in theory, TruffleSqueak shows it in practice and provides more reason to further improve the Graal compiler in the future.

Moreover, we show that TruffleSqueak implements all API requirements of our approach except the optional features of exploratory programming, which is one of its limitations. This and other limitations, such as support for cross-language snapshotting and debugging, require additional research and engineering efforts and are left for future work.

As our fourth contribution, TruffleSqueak demonstrates that our approach is feasible to implement. In addition, it shows that there is a lot to be learned just from supporting a self-sustaining programming system on a polyglot VM, for example, in terms of the VM's performance characteristics or the design decisions made by its language implementation framework.

12. Case Studies Based on TruffleSqueak

As an exploratory tool-building platform, TruffleSqueak can be used by four types of developers working with GraalVM for different purposes: Tool developers can build on existing tools and rapidly create new ones, for example, to support developers in writing polyglot applications. TruffleSqueak allows application developers to build polyglot applications at run-time. It also makes dynamic run-time data explorable, which can help these developers to understand how different languages interact with each other and how they can be combined effectively. This capability of TruffleSqueak can also be used by language and runtime developers to explore GraalVM internals while they are running.

This chapter presents five different case studies based on TruffleSqueak that give concrete examples of how different developers can use and benefit from it. For building these examples, we used the exploratory tools from TruffleSqueak and its ability to build tools and applications at run-time. Each case study starts with a research question followed by a short problem statement, a description of the project outcome, a selected implementation detail, and finally a summary of insights and lessons learned.

12.1. Building a Polyglot Notebook System

> **Research Question** How could polyglot VMs enable and support data scientists and researchers to use multiple programming languages within a single notebook?

Problem Statement Computational notebook systems, such as Jupyter [88] and Google Colab [56], have become popular tools for data scientists and researchers in recent years. Inspired by Literate Programming [89], notebooks allow their users to combine code with text, for example, to document instructions, to explain algorithms, or to discuss results. For this, users can typically add two types of cells to a notebook: text cells and code cells. Code cells can usually be evaluated by a notebook kernel, a server that often runs on a remote, high spec machine or cluster to allow computationally intensive workloads.

Evaluation results of code cells are then displayed in corresponding output cells, which often support different MIME media types and can therefore provide different types of visualizations.

Since the scientific community uses many different programming languages, especially in fields such as data analysis and machine learning, notebook systems usually support multiple languages, often by allowing users to connect to kernels for different languages. The Jupyter project, for example, started with support for Julia, Python, and R and today supports many other languages through over 150 different kernels [152]. Standard notebooks, however, are usually limited to a single programming language. To build data processing pipelines with more than one language, files and databases are often used to exchange data between languages. Polyglot notebook systems, such as SoS Notebook [148] and BeakerX [206], demonstrate that there is a need for being able to use multiple languages in the same notebook. These systems, however, usually use IPC and orchestrate and combine multiple kernels for different languages.

Large data sets, complex objects, and frequent interactions between languages can, however, pose problems for these systems. Polyglot VMs, on the other hand, can avoid data duplication, data synchronization, and other disadvantages of common language integration techniques. In this case study, we built a polyglot notebook system in TruffleSqueak and based on GraalVM.

Project Outcome Initially, TruffleSqueak's `PolyglotWorkspace` tool allowed us to mimic a very basic notebook experience without any further work: For each notebook cell, we could open a new instance of the tool and select a particular language. And the shared polyglot bindings object could be used as a global namespace for sharing objects between languages.

We then built a new `PolyglotNotebook` tool that allows chaining of the same text editors that are also used in the workspace tool. This means that they also support syntax highlighting for different languages through the Rouge library from Ruby, which we presented in Section 10.1. Since it is important to know what is shared between languages, we further embedded our polyglot object explorer in the sidebar of the `PolyglotNotebook`. To complete the first version of the notebook system, we added some controls for adding, removing, re-arranging, and running code cells, for selecting the language of a code cell, as well as for loading and saving the notebook. Since then, we have explored different ideas and extended our polyglot notebooks in different ways.

As a data analysis example, assume we are interested in the number of contributors per country of a particular conference. For this, we need to take care of mainly three tasks: First, we need to download the list of all contributors from the conference website and extract the data from the HTML

file. Second, we need to perform some form of data cleansing because the data may be incomplete or inconsistent. From that, we can derive a list of countries. And lastly, the list needs to be aggregated and visualized, for example, with a plot.

Although all these three steps can be done in one language, different languages may suit one task better than others. The `PolyglotNotebook`, on the other hand, allows us to choose a different language for each task. Figure 12.1 shows the notebook we created for our data analysis example. On the left, there are three text cells and three code cells with corresponding output cells. The sidebar on the right displays the notebook's bindings object. While the text cells document the analysis task using the Markdown format, the three code cells make use of three different languages:

Ruby code cell (red) The `nokogiri` parsing library from Ruby makes it easy to download and parse an HTML file. In this case, we use this library to extract the affiliation column and the country column from the contributor list of the conference website. The result, a Ruby array of arrays of strings, is stored in the notebook's bindings object under the name "rows".

Python code cell (blue) The `pycountry` library written in Python provides access to a list of all countries. With this library, we search for country names within the Ruby array using Python list comprehension. The result of the cell is a Python list of country names and stored under the name "countries" in the notebook's bindings object. The explorer on the right can be used to explore the contents of the notebook's bindings object including the elements of the Python list and its interoperability members.

R code cell (gray) Finally, we wrap this list in an R data frame object and use the `aggregate()` builtin to aggregate the countries before passing it into `ggplot2`, an R package for data visualization. The `%ggplot2` notebook magic command at the beginning of the cell instructs our notebook system to use an `RPlotMorph`, a polyglot UI component we presented in Section 10.2, for rendering the resulting plot within the output cell.

The `PolyglotNotebook` tool provides different features that we now explain in more detail. The most noticeable feature is that code cells highlight the selected language with colors and an icon of the language. This helps users to keep track of the language they use throughout their notebooks. Instead of Truffle's polyglot bindings object, each notebook uses an instance-specific `PNBKeyValueStore` object for sharing objects between languages, to avoid polluting the global bindings namespace and interferences with other notebook instances. This `PNBKeyValueStore` makes use of TruffleSqueak's language-level implementation of the language interoperability protocol from GraalVM

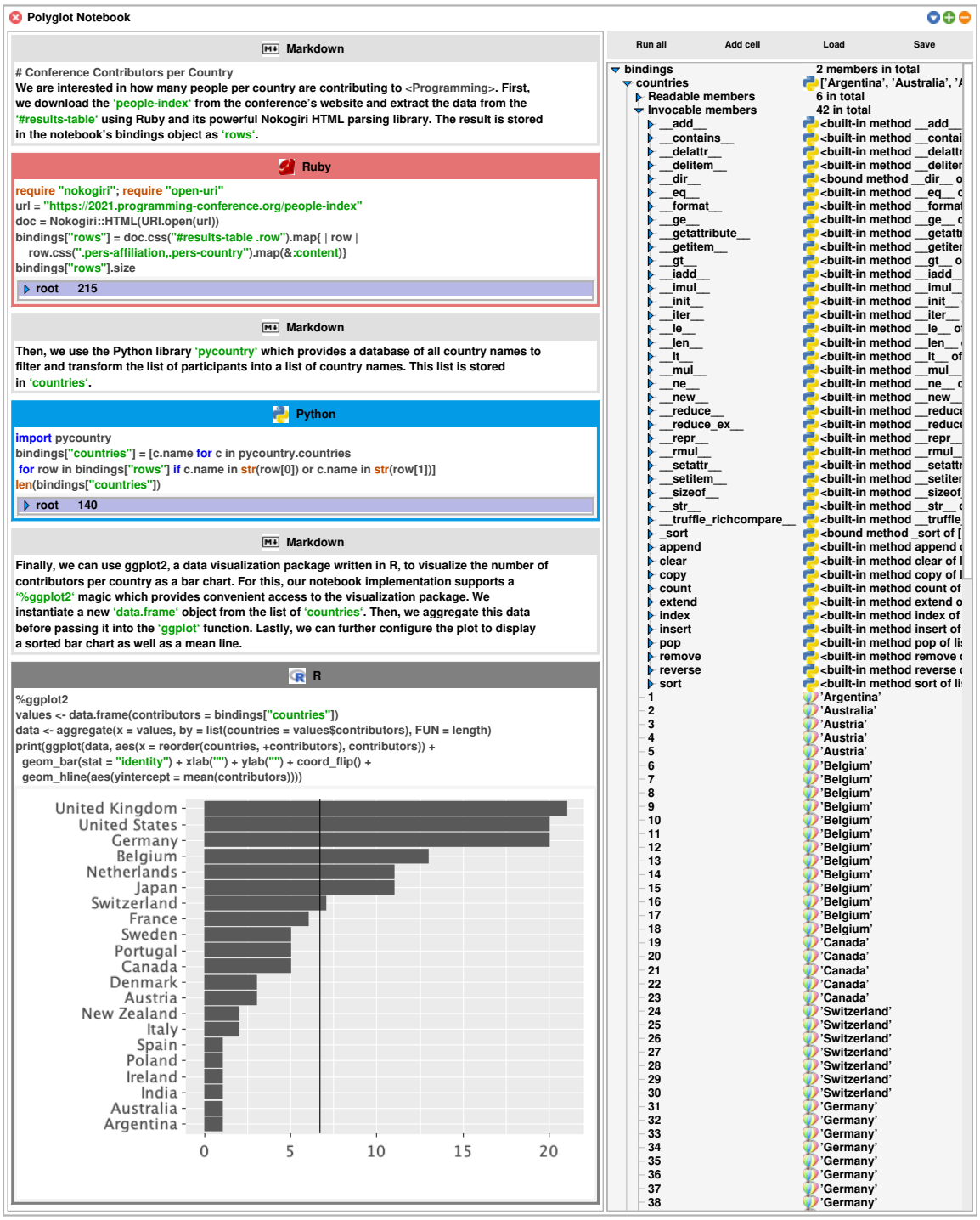


Figure 12.1.: A polyglot notebook example visualizing the number of contributors per country of the ‹Programming› 2021 conference using Ruby, Python, R, and a tool built in Squeak/Smalltalk.

Listing 12.1: Simplified implementation of the `PNBCodeCellContainer class>>isValidNBJson:` utility used by `PolyglotNotebook` for validating saved notebooks with `nbformat`.

```
isValidNBJson: aJsonString
  isValidateNBJson ifNil: [
    [ isValidateNBJson := Polyglot eval: #python string: 'import nbformat
def is_valid_nb_json(nb_json):
  try:
    nbformat.validate(nbformat.reads(nb_json, 4)) # use version 4
    return True
  except:
    return False
is_valid_nb_json' ] on: Error do: [ :e |
      self error: 'Failed to load nbformat. Install via `pip install
      ↪ nbformat`.'.
      ^ false ] ].
  ^ isValidateNBJson value: aJsonString
```

to control its appearance in other languages. For this, it remaps the member trait of the protocol onto a Smalltalk dictionary that it internally uses. It does this for two reasons: First, to disallow other languages to access its Smalltalk methods and instance variables, and second and more importantly, to ensure that languages can only use strings as keys. An option for the object explorer in the sidebar can further be used to add the top scopes of all languages to the explorer's tree. This allows users to explore global modules, classes, functions, and other components provided by different languages. Furthermore, `PolyglotNotebook` can load and save notebooks in the Jupyter notebook format, which allows sharing across notebook systems.

Selected Implementation Detail The official Jupyter notebook format is based on the JavaScript Object Notation (JSON) format. For this reason, `PolyglotNotebook` uses a JSON implementation written in Smalltalk for loading and saving notebook files. This makes it possible to use the notebook system even when it is not running on GraalVM, but then it is limited to Smalltalk code cells. More importantly, the notebook format supports per-notebook and per-cell metadata, which our system uses to persist the selected language per code cell. To ensure that the JSON output of our tool can be read by other notebook systems, we have added a validation step to the saving process. This validation step is only triggered if Python is supported and checks that the JSON output conforms to the official notebook format schema. This is done through an integration of `nbformat`, the reference implementation of the Jupyter notebook format written in Python.

Listing 12.1 contains the code of a utility method used for validating saved notebooks with `nbformat`. The method is implemented on the class side and uses the `isValidateNBJson` class variable for referencing a Python method. The lazy-initialization pattern makes sure that the variable is always correctly

initialized. This happens when the method is used for the first time. But because Python objects are not persisted as part of Smalltalk images, the variable can also be `nil` after an image is loaded. The Python code evaluated through TruffleSqueak's polyglot API tries to import the `nbformat` module. It then creates a helper method that reads a JSON string in version 4 of the file format and calls the `validate()` function on the result. If any of this fails with an exception, the JSON does not conform to the Jupyter notebook format. Therefore, the helper returns `true` if validation is successful, and `false` otherwise. If the import statement fails, however, an error is thrown. In this case, the utility method informs the user about the error and suggests installing `nbformat`. Otherwise, the polyglot API returns the result of the last line of the Python code, which is the helper method. According to GraalVM's language interoperability protocol, Python methods are executable. In TruffleSqueak, executables can be invoked with one argument through the `value:` message of Smalltalk `BlockClosures`.

Lastly, the integration of `nbformat` is an example of how polyglot programming can be applied in tools. With a few lines of code, the output of a tool can be validated with the reference implementation of a specific file format to ensure compatibility.

Insights and Lessons Learned This project demonstrates that the ability to compose existing tools to build new ones at run-time makes TruffleSqueak a useful tool-building platform. Moreover, its exploratory tools are suitable for data analysis, which in itself is often exploratory [205].

In terms of sharing objects between languages, the system allows us to explore different ideas: The example from Figure 12.1 shows that objects can be exchanged explicitly through a bindings object. Initially, we used Truffle's polyglot bindings object for this. But because it is a global namespace without any support for scoping, we decided to use a dedicated key-value store per notebook instance. This avoids name clashes and interferences with other notebooks and applications that use the global polyglot bindings object in some way.

Furthermore, we also experimented with approaches to automate sharing of variables between languages. Whenever a code cell is executed, the tool takes a snapshot of all keys available in the top scope of the corresponding language before the execution. After the execution, it compares these keys against the current set of keys to identify newly introduced variables. The tool keeps track of all variables and their languages introduced by all code cells of a notebook. Based on this information, it can copy corresponding objects from one top scope into the top scope of another language before a code cell is executed. This way, it is possible to automatically share variables instead of

having to explicitly use the bindings object. This approach, however, has some disadvantages and relies on some requirements that all GraalVM languages must fulfill: First, each language must provide a top scope, which is currently an optional feature. Second, local variables defined as part of code evaluated through the polyglot API must be stored in the top scope. Moreover, variable names may clash with the elements in the top scopes or the keywords of other languages. A local variable named `import`, for example, cannot be used in Python because `import` is a keyword of the language. Another requirement is that top scopes are writable, which some languages may not support. Instead of copying values across top scopes, the tool could also pass in the variables through a local scope when evaluating code. The support for this in Truffle, however, is still experimental and inconsistently implemented across different languages. A third option is to generate import statements for the current language and prepend them to a code evaluation request. This, however, requires the tool to have knowledge about all languages. This means that the tool is no longer language-agnostic and therefore no longer works for newly added languages automatically.

Furthermore, we realized that the `PolyglotNotebook` can also be a helpful tool for language and runtime developers. Through introspection, it is easy to analyze language internals or GraalVM internals and to create visualizations. As an example, we analyzed the number of slots of classes and their types across Squeak/Smalltalk at run-time to determine reasonable numbers for inline fields in TruffleSqueak's object layout (see Figure C.1). In another example, we used a notebook to analyze the compilation queue of the dynamic Graal compiler (see Figure C.2).

TruffleSqueak allowed us to better understand the requirements for a notebook system built on top of GraalVM. In Section 13.2, we show that the insights gained from this project are not specific to TruffleSqueak and can also be applied to other notebook implementations, such as an actual Jupyter kernel or the notebook system of VS Code.

12.2. Adding Support for Polyglot APIs to Code Editors

> **Research Question** How could code editors support developers in using the polyglot APIs from different GraalVM languages?

Problem Statement To enable polyglot programming, GraalVM languages provide access to other languages through polyglot APIs. Table 12.1 gives an overview of the core polyglot APIs of five different GraalVM languages. Each of them allows strings and files to be evaluated for a given language. In

Table 12.1.: Overview of the core polyglot APIs provided by five different GraalVM languages. Although all of them provide means to evaluate strings and files as well as to export and import values to and from Truffle's bindings object, the APIs are different across languages.

GraalVM Language	Core Polyglot API

FastR	`eval.polyglot(id, string)`
	`eval.polyglot(id, path=path)`
	`export(name, value)`
	`import(name)`

Graal.js	`Polyglot.eval(id, string)`
	`Polyglot.evalFile(id, path)`
	`Polyglot.export(name, value)`
	`Polyglot.import(name)`

GraalPython	`import polyglot  # import the module before use`
	`polyglot.eval(string=string, language=id)`
	`polyglot.eval(path=path, language=id)`
	`polyglot.export_value(value, name=name)`
	`polyglot.import_value(name)`

TruffleRuby	`Polyglot.eval(id, string)`
	`Polyglot.eval_file(id, path)`
	`Polyglot.export(name, value)`
	`Polyglot.import(name)`

TruffleSqueak	`Polyglot eval: id string: string.`
	`Polyglot eval: id file: path.`
	`Polyglot export: name value: value.`
	`Polyglot import: name.`

addition, objects can be shared between languages through an import/export mechanism. Although they support the same set of core features, the way these features can be used differs across languages. Graal.js, TruffleRuby, and TruffleSqueak, for example, provide a dedicated, preloaded `Polyglot` module or class. In GraalPython, on the other hand, the `polyglot` module needs to be explicitly imported before use. It further needs to avoid a name clash with Python's `import` keyword, which is why the import/export functions are called `import_value()` and `export_value()`. FastR provides its polyglot API through a set of builtins instead. Moreover, the `eval` functions of FastR and GraalPython support strings and files through keyword arguments. The other three languages provide two functions, one for evaluating strings and one for files. Furthermore, each polyglot API follows the established naming conventions of the corresponding language, such as camel case or snake case.

From the perspective of developers, these differences in the APIs of GraalVM languages are problematic. Having to think about the correct usage of each API adds additional cognitive overhead for developers and therefore increases the potential for errors. Appropriate suggestions for code comple-

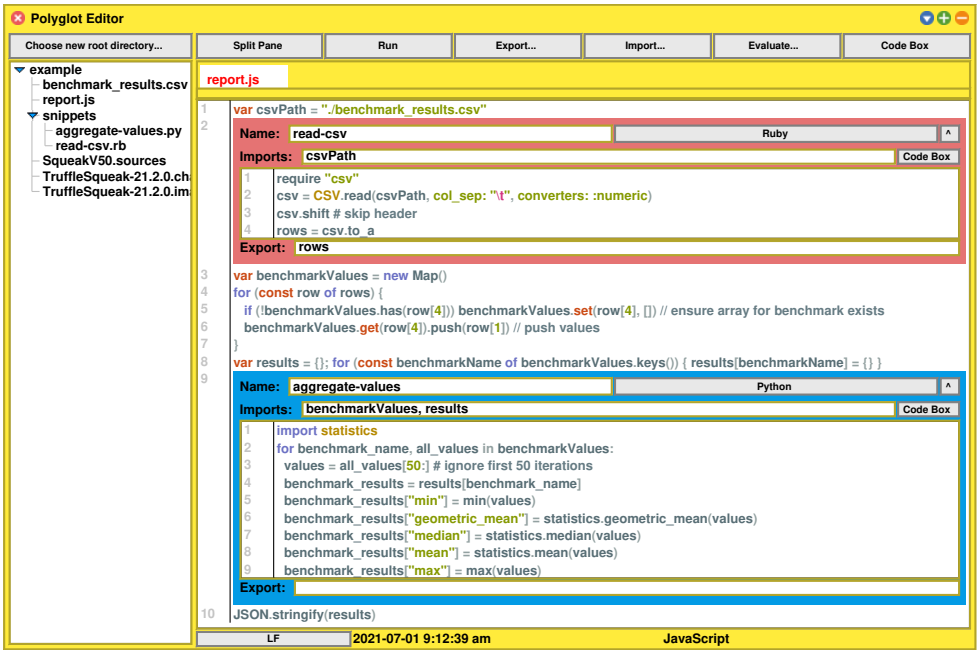


Figure 12.2.: A screenshot of TruffleSqueak's `PolyglotEditor` demonstrating its support for code boxes.

tion, as provided by GraalVM's extension for VS Code [137], can help to avoid the worst case of having to refer to the documentation. The correct name for an API, however, can be hard to discover through code completion. Think about FastR's builtins for example. More importantly, the polyglot APIs still add additional cognitive overhead and additional work for developers: They must keep track of all exported and imported values and maintain a correct order within their polyglot application. Also, they might want to structure code of different languages in separate files, which they have to manage. In this case study, we explored how code editors can support developers in using the polyglot APIs of GraalVM languages beyond code completion.

Project Outcome Although Squeak/Smalltalk code is managed in objects, not in files, a file-based code editor was easy to build in TruffleSqueak. Similar to the `PolyglotNotebook` tool, TruffleSqueak's `PolyglotEditor` is composed of existing components from Squeak/Smalltalk, such as a tree view for visualizing a folder structure on the file system and the text editor with polyglot support for syntax highlighting.

Figure 12.2 shows a screenshot of the `PolyglotEditor` tool. The sidebar on the left visualizes a selected folder structure. From there, files can be opened within tabs underneath the top bar that provides different controls. The "Split Pane" button allows users to open two files side by side. "Run"

evaluates the currently opened file and opens an inspector on the result of the execution. The "Export..." button opens a dialog that asks the user for a name, followed by a dialog asking for a value. For a given name-value pair, the editor inserts an export statement for the language of the current file at the current position of the cursor. Furthermore, it keeps track of the exported names. The "Import..." button opens a list that consists of both, the names of values that have been exported as part of this editor instance as well as the names that are currently present in Truffle's polyglot bindings object. Similar to the previous two buttons, the "Evaluate..." button allows developers to generate a code evaluation statement in the current language. First, it provides a list of all languages that are available in the active GraalVM installation. Developers are then prompted to enter the code they want to evaluate in the selected language. Lastly, the button on the very right inserts a new *code box* at the cursor's position. These code boxes are inspired by language boxes [157] and are editors embedded in an editor that can be set to different languages. Alongside code, each code box has a name and language. Polyglot imports and exports can optionally be set. With this information, the editor creates a file in the `snippets` directory in the current working directory. The file name is derived from the code box's name and the selected language. If imports or exports are defined, the editor generates appropriate import and export statements for the corresponding language and adds them to the file. Lastly, it inserts a polyglot evaluate file request into the parent file of the code box. When creating new code boxes, the editor further offers to re-use existing code boxes if any are found in the `snippets` directory. Moreover, code boxes can be collapsed and expanded by clicking on the small `^` button displayed in the top right corner of each box. This way, foreign code within code cells can be easily hidden from the overall view.

The example opened in the editor shown in Figure 12.2 makes use of two code boxes. Similar to the `PolyglotNotebook` tool, the editor encodes language selections with colors. The `report.js` file opened in the editor creates performance reports in the JSON format for a given CSV file with benchmark results. The benchmark results analyzed in this example are from the Are We Fast Yet (AWFY) benchmarks presented in Appendix B.

The first code box is set to Ruby and loads and parses the CSV file into an array. For this, it imports the `csvPath` variable from JavaScript and exports the Ruby array stored in the `rows` variable. In JavaScript, a map of values per benchmark and a `results` object are created from the `rows` Ruby array. The second code box uses Python and imports both, the map stored in `benchmarkValues` and the `results` object. For each benchmark, the first 50 values are skipped to reduce the effect of warmup in benchmarks. The remaining values are then aggregated with Python's `statistics` module and

Listing 12.2: Excerpt from the actual `report.js` file that was generated as part of the example shown in Figure 12.2.

```
1 var csvPath = "./benchmark_results.csv"
2 // CODE BOX BEGIN:./snippets/read-csv.rb
3 Polyglot.export("csvPath", csvPath)
4 Polyglot.evalFile("ruby", "./snippets/read-csv.rb")
5 var rows = Polyglot.import("rows")
6 // CODE BOX END
7 var benchmarkValues = new Map()
8 // ...
```

the `min` and `max` builtins. Since the Python code box stores aggregated values in the `results` object from JavaScript, it does not need to export anything. Finally, the `results` object is converted into a JSON string with JavaScript's `JSON` module.

When the `report.js` is executed, the result is a JSON-formatted string containing statistical information about the benchmarks, such as the geometric mean per benchmark. The code written in this file could, for example, be used further within a webhook of Node.js server that provides performance results through a REST API.

Note that although this example uses Graal.js' polyglot API to evaluate two files, the API usage is completely hidden from the user. The Ruby and Python files managed by the editor can, however, be found in the `snippets` directory listed as part of the folder structure.

Selected Implementation Detail One of the main features explored in the `PolyglotEditor` project are the code boxes for hiding polyglot evaluate file requests from the users. For this, the editor creates and manages files in the background and generates appropriate polyglot API calls. In the following, we show some of the actual output created by the editor for the example from Figure 12.2.

Listing 12.2 shows the first seven lines of the `report.js` from the example shown in Figure 12.2. The `PolyglotEditor` uses code comments to persists meta-information about code boxes in files. In this case, the code for the Ruby code box is in between lines two and six: Line three exports the `csvPath` variable through the polyglot API. In the next line, the `read-csv.rb` file stored in the `snippets` directory is evaluated. And line five makes the `rows` array from Ruby available in the rest of the `report.js`.

Listing 12.3 shows the contents of this `read-csv.rb` file: The first line imports the `csvPath` variable from JavaScript. Line two contains the begin tag for code boxes as well as the metadata entered through the UI. Lines three to six are the actual lines written in the code box, followed by the end tag for code boxes. And the last line exports the `rows` variable through the polyglot API.

Listing 12.3: Contents of the `read-csv.rb` file generated by the `PolyglotEditor` for the Ruby code box shown in Figure 12.2.

```ruby
csvPath = Polyglot.import("csvPath")
# CODE BEGIN:{"boxName":"read-csv",
↪  "exportVariables":["rows"],"importVariables":["csvPath"],"language":"ruby"}
require "csv"
csv = CSV.read(csvPath, col_sep: "\t", converters: :numeric)
csv.shift # skip header
rows = csv.to_a
# CODE END
Polyglot.export("rows", rows)
```

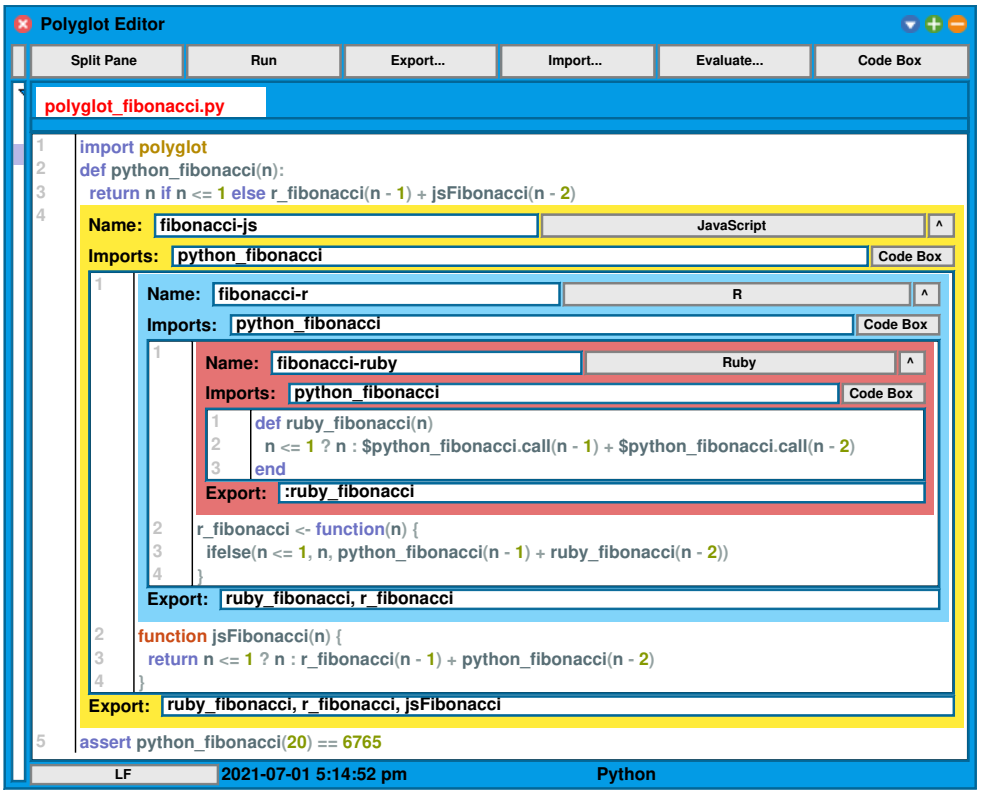


Figure 12.3.: A polyglot, recursive algorithm to calculate Fibonacci numbers using the nested code boxes feature of TruffleSqueak's `PolyglotEditor`.

In terms of the UI, the `PolyglotEditor` makes use of `TextAnchors`. Such a `TextAnchor` can be added as an attribute of the editor's `Text` object to display a `Morph` object at a specified position. Code boxes are therefore implemented as morphs and added to text using `TextAnchors`. This, however, has an interesting side-effect: Since code boxes contain an embedded editor that also has a `Text` object, code boxes can be nested indefinitely.

In Figure 12.3, we have used nested code boxes to implement a polyglot, recursive Fibonacci algorithm. Each implementation of the algorithm in one

language uses the implementation of one or two other languages recursively. The `r_fibonacci` function, for example, uses `python_fibonacci` and `ruby_fibonacci` if `n > 1`. While this is a toy example, the assertion in line five of the main Python file demonstrates that this polyglot implementation can compute correct Fibonacci numbers. More importantly, it shows that the editor generates correct files for nested code boxes across different languages.

Insights and Lessons Learned The `PolyglotEditor` project demonstrates that TruffleSqueak also allows the exploration of ideas for commonly static tools such as code editors and gives examples of how such tools can be enhanced with dynamic run-time data. The basic UI and features of the code editor were easy to build with the `ToolBuilder` infrastructure and the Morphic framework from Squeak/Smalltalk. Furthermore, `TextAnchors` not only made it easy to implement code boxes but also enabled nesting.

Moreover, this project allowed us to better understand the requirements and language specifics for integrating polyglot APIs of GraalVM languages in code editors: Similar to code completion, the `PolyglotEditor` needs to know how to generate statements for evaluating strings and files as well as for exporting and importing values across all languages. In addition, the editor must have an understanding of how assignments are defined in all languages so that it can make imported values available under a given variable name. GraalPython further demonstrated that the editor also needs to know how to manage and extend language imports. Since we decided to manage metadata within the files, as opposed to separate metadata files, the `PolyglotEditor` also knows about how comments work across languages.

Another takeaway from this project is that code editors can allow developers to use polyglot APIs in a consistent way. The steps for creating polyglot evaluate, export, and import requests and for code boxes are always the same, even if developers switch between files written in different languages. As the detection of currently exported values through the polyglot bindings object demonstrates, dynamic run-time data can further enhance the features of the editor with accurate information.

As part of this project, we further experimented with ideas that help developers to switch between languages. Through linter-like annotations, we tried to highlight programming and stylistic errors that often occur during polyglot programming. The first two of such errors we identified are semicolons used or not used at the end of each line as well as the use of parentheses and brackets. Ultimately, we concluded that editors should support linting for the languages in use, similar to syntax highlighting and for example through a combination of linters for different languages.

12.3. Helping Developers to Find Re-Usable Code

> **Research Question** How could we help developers to find appropriate, re-usable code for building polyglot applications?

Problem Statement Many developers are familiar with more than one programming language and polyglot programming allows them to combine their knowledge and thus languages, which increases productivity. Due to many different reasons such as personal preferences, education, or work experience, it is common that developers know some languages better than others, including the libraries, frameworks, and tools that each language provides. Furthermore, many of the underlying programming techniques and concepts are universal across languages. Therefore, experienced developers can often read and understand code of a language, even if they are not familiar with writing code in that language.

A crucial factor with regard to software reuse is to know about and have experience with existing software that can be re-used. Programming with polyglot VMs further allows developers to re-use code across languages. With every additional language that a polyglot VM supports, the libraries and frameworks of an entire language ecosystem are made available for reuse. Additionally, the idea behind polyglot programming is to allow developers to always use the "best" language for the task. But what if developers do not know that some tasks can be implemented much easier in some languages than others? In this case study, we explored ideas that help developers to find re-usable code when building polyglot applications.

Project Outcome Developers often use the internet to search through programming resources and to exchange knowledge with others. StackOverflow, for example, is a popular online platform that developers use to ask and answer programming questions [204]. Many of these programming answers contain code snippets to illustrate possible solutions. An analysis of code snippets from StackOverflow suggests that for natural language texts, some of them even provide running solutions that work out-of-the-box [223].

As part of this project, we designed and implemented a tool that allows developers to search StackOverflow from within their IDE, thus reducing expensive switches between the programming environment and a web browser redundant. Through exploratory programming features, developers can further explore search results interactively and adapt them for their needs.

Figure 12.4 shows a screenshot of the polyglot code finder tool, which can be opened from within a `PolyglotNotebook` or a `PolyglotEditor`. Developers can enter a search query and select the languages that they would like results

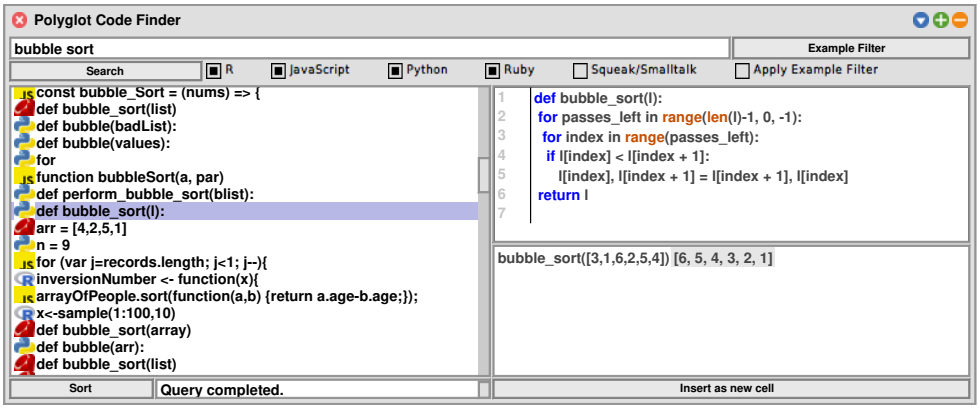


Figure 12.4.: The polyglot code finder allows developers to search for re-usable code snippets from StackOverflow.

for. In this example, the search query is "bubble sort" and the languages R, JavaScript, Python, and Ruby are selected. After clicking the "Search" button, the tool performs the search through StackOverflow's public API. Content can be tagged on StackOverflow and the tool uses this to filter for the selected set of languages. Results appear in the list on the left as a query is running and can be sorted by the score of an answer, the number of lines of code, or by the initial order they were received. The status bar next to the "Sort" button on the bottom informs users about whether a query is currently running or has been completed. Each item in the list shows an icon of the snippet's language as well as the first line. When users select an item, the corresponding code snippet is shown in the upper text editor on the right. In this case, it shows an implementation of a bubble sort algorithm written in Python. Users can adjust code snippets for their needs and use the embedded workspace underneath the editor to interactively evaluate code to try out the snippet. Once users are happy with a snippet, they can insert it directly into the corresponding `PolyglotNotebook` or `PolyglotEditor` that opened the tool.

Figure 12.5 shows the code cell added to a `PolyglotNotebook` by the code finder tool. In addition to the code snippet, the cell also includes information on the StackOverflow source, the author, as well as the license as a comment. In a `PolyglotEditor`, the tool inserts an appropriate code box in the currently opened file and at the cursor's position.

Inspired by the `MethodFinder` [187], a Squeak/Smalltalk tool that helps users to find methods for a given pair of input and output objects, the code finder supports example filters. An example filter, which consists of a list of input objects as well as an expected output object, can be set through the button on the top right corner of the tool. When the filter is applied through the corresponding radio box, the code finder will try to find snippets

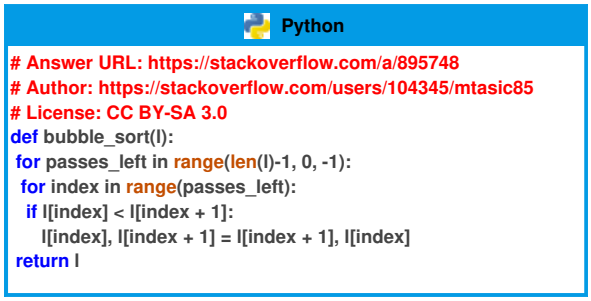


Figure 12.5.: A code cell added through the polyglot code finder from Figure 12.4 contains the code snippet as well as information about the original StackOverflow source, the author, as well as the license.

Listing 12.4: Simplified implementation of a `performSearchAction` method that is invoked when users click on the code finder's "Search" button.

```
1  performSearchAction
2    rubyQuery := Polyglot eval: #ruby string: self newQueryCode
3      names: #('query_string' 'add_snippets_callback')
4      arguments: {self queryString. [ :snippets | self addSnippets: snippets ]}.
5      self updateStatusBar: 'Processing query...'.
6      rubyQuery start.
7    [
8      [ (Delay forSeconds: 2) wait. rubyQuery poll ] whileTrue.
9      self updateStatusBar: 'Query completed.'.
10   ] fork.
```

that correctly transform the input objects into the output object. For this, it evaluates each code snippet with the given input and rejects snippets that fail or that do not produce the expected result. Since code snippets can potentially contain harmful instructions, the tool can make use of GraalVM's sandboxing feature designed to run untrusted code, for example, by restricting access to the file system. Depending on the number of snippets, this process can take some time. The remaining items in the code snippet list, however, produce the correct output for the given input objects and may therefore be good candidates for reuse.

Selected Implementation Detail The polyglot code finder is written in a polyglot way from the very start. Its UI is built using `ToolBuilder` from Squeak/Smalltalk. For performing and extracting code snippets from web requests in parallel, it uses Ruby. And extracted code snippets are managed in Python objects.

Listing 12.4 shows how the method that initiates a new code search is conceptually implemented. The expression from lines two to four evaluates Ruby code that allocates a new `StackOverflowQuery` instance initialized with the current query string and an `add_snippets_callback` for adding new

snippets to the list of snippets displayed in the tool's UI. The fifth line updates the status bar of the tool. The following line instructs the Ruby object to start querying StackOverflow. Internally, the object starts multiple, OS-level threads for downloading result pages and as well as answer pages in parallel. It then parses the JSON data from StackOverflow using Ruby's `json` module and the `nokogiri` parsing library to extract code snippets from HTML `code` blocks found in the HTML body of each answer. All code snippets found are then stored in objects of a Python model. The `BlockClosure` from lines seven to ten finally spawns a new Smalltalk process that periodically updates the UI with newly found snippets. For this, the process runs another `BlockClosure` that waits for two seconds and then invokes the `poll` method of the `rubyQuery` object until it returns `false`. Within this method, the `rubyQuery` invokes the `add_snippets_callback` with all snippets found since the last invocation of the method and signals whether the query is still running by return `true`. Once the execution of the query has been completed, `poll` returns `false` and line nine updates the status bar again, before the Smalltalk process completes.

Insights and Lessons Learned This project is an example of how new tooling ideas for polyglot programming can be explored and implemented in a polyglot way with TruffleSqueak. Moreover, the tool demonstrates that new tools can easily be integrated into other TruffleSqueak tools, such as the polyglot notebook or editor tools.

Depending on the query, StackOverflow can provide multiple search result pages as well as answer pages for each listed question. All of which need to be requested individually by our tool. For speeding up the querying process and to run it independently from the UI, we wanted to use OS-level threads. While Squeak/Smalltalk only supports green threads, polyglot programming allowed us to use other languages that support OS-level threading. Initially, we thought we could use Python for this. But after creating a model for code snippets, we realized that GraalPython, at that time, neither supported the `urllib` module for querying the StackOverflow API nor the `threading` module. TruffleRuby, on the other hand, did support both Ruby's `Net::HTTP` client and the `concurrent-ruby` gem for threading. Instead of porting the model we had written in Python to Ruby, polyglot programming further allowed us to use the existing Python model from within Ruby. The use of Ruby is therefore incidental and due to lacking support for modules in GraalPython. Nonetheless, it is not uncommon that during the implementation process, something turns out to be impractical for other reasons than runtime compatibility. Hence, this is an example of how polyglot programming provides developers with more flexibility: They can try out different implementation strategies across multiple languages without having to port existing code.

Furthermore, we made a few interesting observations with regard to code snippets that our tool finds on StackOverflow: The number of code snippets is different across languages, which makes sense considering that language communities are different in size and differently represented on StackOverflow. Another reason for this could be that some languages are indeed used more often for specific tasks than others. In the example shown in Figure 12.4, the number of code snippets written in R was noticeably lower than the ones found for JavaScript, Python, and Ruby. The example also shows another observation that we made: Code snippets shared on StackOverflow do not always contain pure functions. Sometimes the code is unstructured, taken from an interactive shell session, or contains a class definition. Although developers can oftentimes adapt such code easily, it poses a problem for our example filters as code evaluation may fail due to a compilation error or because the wrong object is returned. In the future, the tool could further be connected to code hosting platforms such as GitHub that may provide more structured code. Another observation that we made is that the language version plays an important role: Many results for Python, for example, were written in Python 2. GraalPython, on the other hand, only supports a specific version of Python 3. Other GraalVM languages, such TruffleRuby or FastR, also only support one specific version of the language they implement.

12.4. Understanding Run-Time Behavior of the Graal Compiler

> **Research Question** How could we help runtime and language developers to better understand the complex behavior of the Graal compiler at run-time?

Problem Statement The Graal compiler is a modern JIT compiler for Java as well as for languages implemented in the Truffle framework. As such, it performs state-of-the-art performance optimizations and interacts with other complex components of the JVM. The main UI-based tool for the compiler is GraalVM's Ideal Graph Visualizer, which can be used to inspect compiler graphs interactively and after they were dumped. Most other tools for language and runtime developers are command-line based and usually designed for one specific task, such as logging the behavior of specific performance optimizations. Sometimes, however, the interaction of different components and optimization strategies are responsible for specific behavior. Warmup behavior, for example, plays an important role and can be influenced in many ways [7].

An instance of GraalVM, however, needs to be restarted whenever developers want to change the set of command-line flags and tools. A restart always causes run-time state to be lost, which is especially costly when a specific state is hard if not even impossible to reproduce. In this case study, we used TruffleSqueak's VM introspection capabilities and built three tools that allow exploration of the Graal compiler at run-time.

Project Outcome Unlike other GraalVM languages, TruffleSqueak supports a full-fledged programming system self-sustained on the user application level. This has two interesting consequences: First, the Graal compiler can significantly influence the responsiveness of TruffleSqueak's programming system. If warmup, for example, is slow, it takes more time for the programming system to become usable. And second, a key assumption of partial evaluation as performed by the Graal compiler is that code stabilizes over time. In TruffleSqueak, however, new tools and applications can not only be opened at any point in time, opened tools and applications can also be changed at run-time. For this reason, we built a tool for monitoring the Graal compilation queue from within TruffleSqueak.

Figure 12.6 shows a screenshot of the TruffleSqueak programming system demoing the `GraalInfoMorph`. In this instance of TruffleSqueak, the polyglot notebook from Figure 12.1 is opened and was fully executed. We also disabled multi-tier compilation and the traversing compilation queue option in the Graal compiler and enabled its throughput mode to simulate its default behavior in GraalVM 21.1.0 and earlier, which often led to large compilation queues in TruffleSqueak. We dropped the `GraalInfoMorph` into the middle of the menu bar on the top to display the current size of Graal's compilation queue. As shown in Figure 10.4, this information is easy to access through the `TruffleRuntime` object. Depending on the current size of the queue, the text is either gray, orange, or red. The thresholds for this as well as the update interval can be adjusted easily by changing its implementation at run-time. More importantly, the `GraalInfoMorph` keeps track of the sampled values over time using a collection. A click on the morph opens an `RLivePlotMorph` on this collection. The `RLivePlotMorph` is a subclass of the `RPlotMorph` presented in Section 10.2 with the additional ability to redraw plots derived from a specific collection periodically. In this case, the morph visualizes the sampled values in a scatterplot and updates the plot every two seconds. The screenshot in Figure 12.6 demonstrates that the polyglot notebook from Figure 12.1 fills the compilation queue with more than 1500 compilation tasks quickly. On the one hand, this is because Ruby, Python, and R, alongside Smalltalk, are being loaded into the GraalVM process including their standard libraries. This makes a lot of new code visible to the Graal compiler. At the same time,

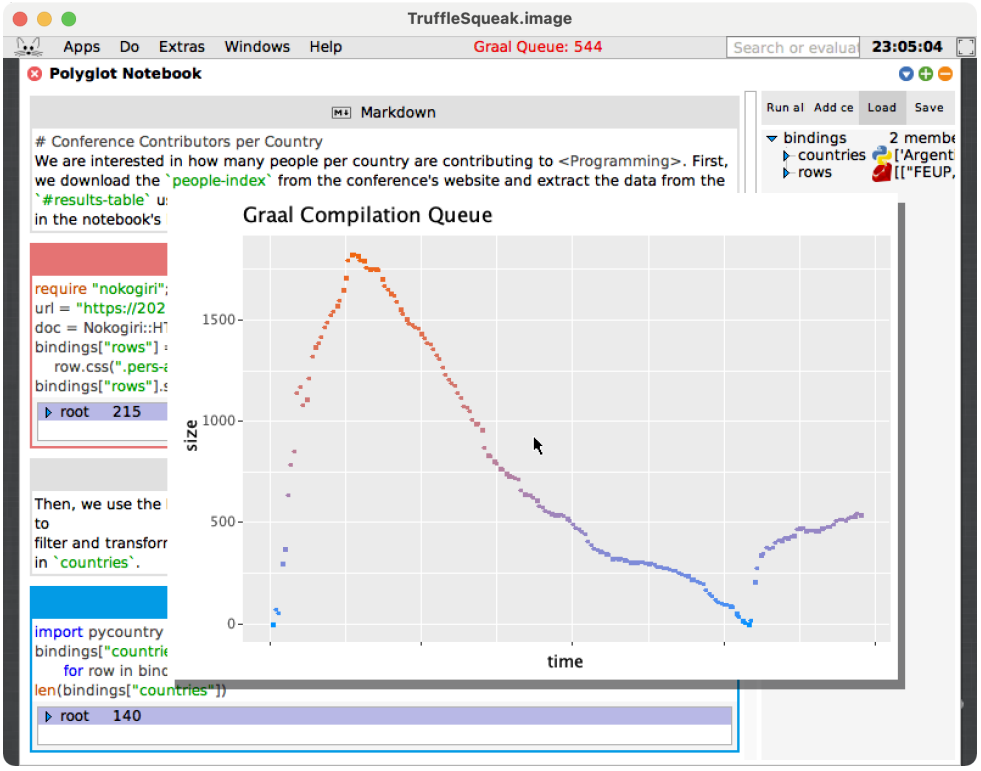


Figure 12.6.: Screenshot of the TruffleSqueak programming system showing the polyglot notebook from Figure 12.1, the `GraalInfoMorph` in the middle of the menu bar, and an `RLivePlotMorph` visualizing the size of the Graal compilation queue over time.

more Smalltalk code is also being executed as we interact with the notebook system. For demonstration purposes, we stopped interacting with the system after evaluating the full notebook. After that, the number of tasks in the compilation queue decreased somewhat steadily. Shortly after the compilation queue was emptied, we clicked on the `GraalInfoMorph`, which again added more compilation tasks to the queue as the opened `RLivePlotMorph` warmed up. This is a good example of a characteristic of self-sustaining programming systems to be aware of: Since everything runs on the level of user applications, TruffleSqueak can measure itself. This can be useful or counterproductive depending on the use case. During performance benchmarking, for example, such behavior can result in undesired noise and inaccuracies of measurement and need to be considered carefully. As the following example shows, this capability can also be used to learn more about the system itself at run-time.

Figure 12.7 shows a screenshot of the `CallTargetBrowser` tool. It is a direct subclass of the `Browser`, the main tool for navigating through the classes of Squeak/Smalltalk and for reading and writing code. As its name suggests,

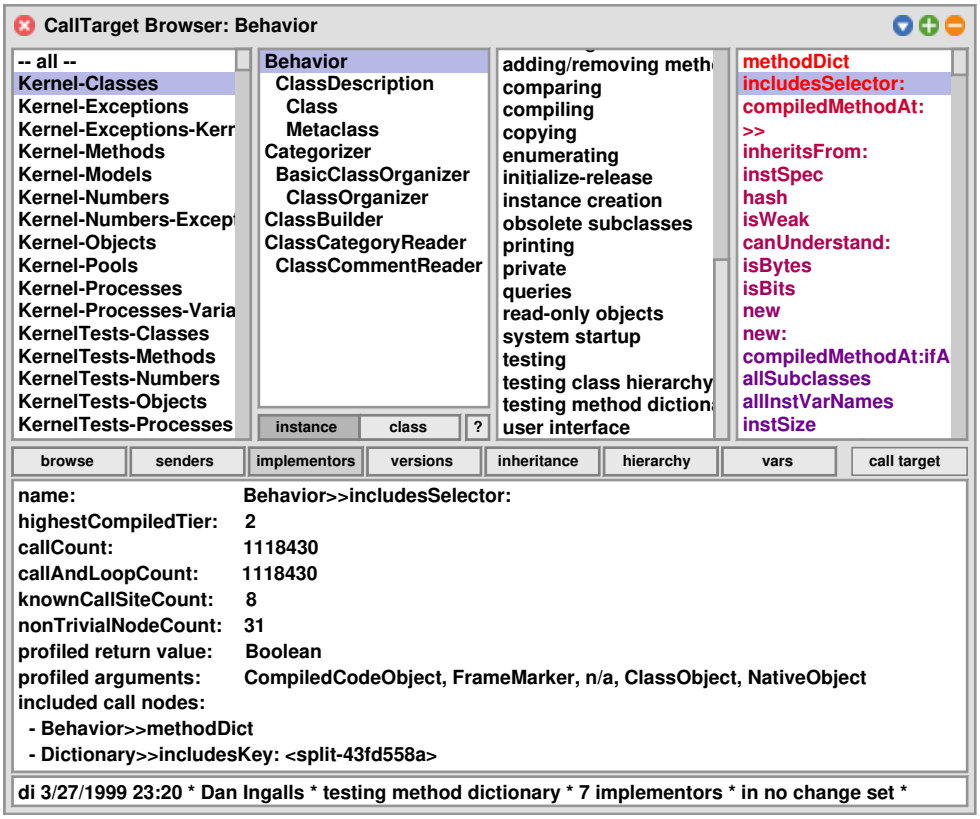


Figure 12.7.: The `CallTargetBrowser` integrates information on the `CallTarget` objects for Smalltalk methods into the `Browser` tool of Squeak/Smalltalk. The list of methods of a class can be sorted by different criteria, for example, by the `callAndLoopCount` or the `nonTrivialNodeCount`. In the lower pane of the `CallTargetBrowser`, users can switch from the source code and bytecode views to a new view that gives detailed information on the call target of the selected method.

Listing 12.5: Implementation of `CompiledMethod>>callTarget`, an extension method added by TruffleSqueak that provides access to the `CallTarget` object of a Smalltalk method.

```
1 callTarget
2   ^ self vmObject ifNotNil: [ :c | c callTarget ]
```

the `CallTargetBrowser` incorporates additional information on `CallTarget` objects into the normal browser. These `CallTargets` are part of Truffle and must be used by language developers to represent methods and other callable objects within their language implementation. They commonly have a direct reference to the AST, often to an AST's `RootNode`. The Graal compiler collects profiling information for `CallTargets` and produces machine code for the ASTs of the `CallTargets` it has decided to JIT compile. Some of this profiling information is directly stored within call targets. Therefore, method objects and call targets are usually connected within Truffle language implementations.

In Squeak/Smalltalk, methods are represented by `CompiledCode` objects, which TruffleSqueak manages in instances of `CompiledCodeObject` of its object model. TruffleSqueak's VM introspection infrastructure, which we presented in Section 10.3, allows the `CallTargetBrowser` to easily access the `CallTarget` object for any Smalltalk method through the extension method shown in Listing 12.5. Since this infrastructure uses `JavaObjectWrappers` to provide unrestricted access to host Java objects, call targets of Smalltalk methods can be accessed without having to modify TruffleSqueak's language implementation.

The `CallTargetBrowser` uses information from `CallTargets` in two ways: As the screenshot from Figure 12.7 shows, the method list on the right of the tool can be sorted by different criteria, such as the `callAndLoopCount` or the `nonTrivialNodeCount`, of the corresponding `CallTargets`. The extended browser further uses colors to encode the values of the selected criteria relative to each other with colors ranging from red (high) to blue (low). Since `CallTargets` are lazily initialized within TruffleSqueak, Smalltalk methods that have not been executed yet do not have a `CallTarget` object. These methods are listed below the blue ones and colored in gray. In addition, the code pane of the `CallTargetBrowser` provides an additional view for `CallTargets`. Developers can therefore easily switch from existing views, such as to view the source code or bytecodes of a method, to the new view that reveals detailed information on the corresponding call target. In the example shown in Figure 12.7, this view is opened on the `Behavior>>includesSelector:` method. The view shows that at the time TruffleSqueak was running, the Graal compiler has profiled more than one million calls to this method. As a result of the high call count, the `highestCompiledTier` value confirms

that the Graal compiler has compiled this method in its second tier that needs more time than tier one but optimizes code more aggressively. A `highestCompiledTier` value of zero, on the other hand, indicates that a method has not been JIT-compiled at all. Since the method does not contain a loop, the values for `callCount` and `callAndLoopCount` are identical. Furthermore, the Graal compiler also identified eight known call sites. According to Truffle's cost model, the call target has a `nonTrivialNodeCount` of 31. Additionally, the compiler has profiled the types of the return value and the arguments used to invoke this method on the language implementation level. So far, the selected Smalltalk method has always returned a Java `Boolean`. The list of profiles arguments reveals the `Frame` layout used by TruffleSqueak: The `CompiledCodeObject` of the method is always passed in as the first argument. This argument is followed by either a `FrameMarker` or a `ContextObject` that represents the sender or the `NilObject` if the invocation has none. The third argument references a `BlockClosureObject` or `null` if none exists. The fourth argument is always the receiver, a `ClassObject` in this case because the method is implemented on `Behavior`. The last argument is therefore the first argument passed into the Smalltalk method, which so far has always been a `NativeObject`. `NativeObjects` are used in TruffleSqueak to represent Smalltalk `Symbols` among others. Lastly, the call target for `Behavior>>includesSelector:` includes other `TruffleCallNodes`. This is the case if the Graal compiler has decided to inline other methods into a call target. The implementation of `Behavior>>includesSelector:` includes exactly two sends, both of which have been inlined by the compiler. `Behavior>>methodDict` looks up the `methodDict`. `Dictionary>>includesKey:` then checks whether the `methodDict` contains the `Symbol` provided as an argument. The name of this call node has been extended with a "<split>" suffix to hint at another compiler optimization called splitting [141]. This optimization aims at reducing polymorphism in code, which can lead to more efficient machine code. In this case, for example, the splitting heuristic has decided that it makes sense to inline a split version of `Dictionary>>includesKey:`, that is a copy of the original call target for `Dictionary>>includesKey:` allowing to be re-profiled in the context of `Behavior>>includesSelector:`. Furthermore, the `Browser` from Squeak/Smalltalk can update itself periodically to always provide an accurate view of the system. The `CallTargetBrowser` builds on this live feedback mechanism. The method list, for example, is updated periodically allowing developers to see how the `CallTargets` of Smalltalk methods change over time. Similarly, the call target view is refreshed, which allows developers to observe how the Graal compiler profiles their code at run-time.

An additional feature of the `CallTargetBrowser` is illustrated in Figure 12.8: The screenshot shows the `Integer>>+` method opened in the tool. Instead of

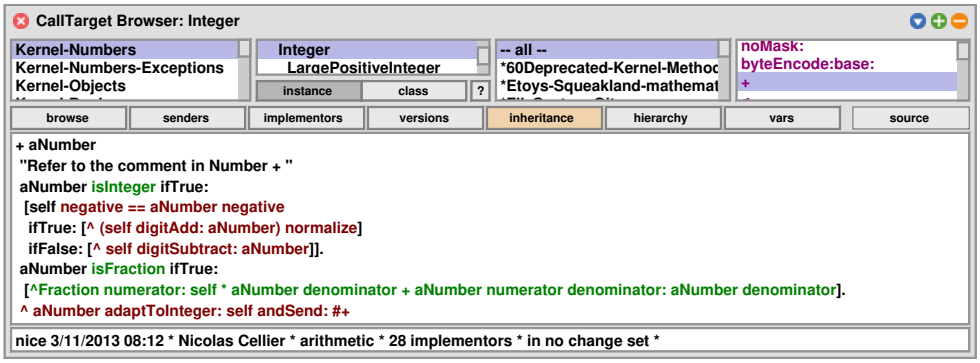


Figure 12.8.: TruffleSqueak's `CallTargetBrowser` can display code coverage information based on Truffle ASTs accessible through call targets.

highlighting the syntax, the code view displays information on code coverage, again periodically updating at run-time. Executed send, return, and store statements are highlighted in green. Red means that they have not been reached yet. Internally, the Java array of bytecode nodes is accessed through the corresponding call target. In TruffleSqueak's language implementation, these bytecode nodes are lazily inserted into the Truffle AST of a method upon first execution of a specific bytecode. To check whether a bytecode has been executed, the tool can therefore simply check whether the array of bytecode nodes contains a node at the corresponding index. Through the `DebuggerMethodMap` of a Squeak/Smalltalk method, bytecodes can be mapped back to source code ranges. This particular example tells us that so far, the `Integer>>+` method has only been called with `Fraction` objects as the `aNumber` argument. This, for example, further implies that the method overrides in `SmallInteger` and `LargePositiveInteger` have either never seen an integer argument, which is unlikely, or that their primitives do not fail to handle an integer argument, which could be a performance problem. Similarly, other methods of the system could be explored to understand in more detail which parts of the codebase are actively being used or not. While this feature comes for free, it is only possible because of an implementation detail in TruffleSqueak's language implementation.

To demonstrate that the general idea of the `CallTargetBrowser` can not only be applied to Squeak/Smalltalk and TruffleSqueak, we have further designed and built `CallTargetBrowserRuby`. This tool allows the exploration of `CallTarget` objects specifically from TruffleRuby. Figure 12.9 shows a screenshot of the tool. The list on the left reveals all constants of Ruby's `Object` class. In this example, the `File` class is selected. The second list, therefore, shows the methods of this `File` class and similarly allows sorting by different call target criteria. For the selected Ruby method, `fnmatch()` in

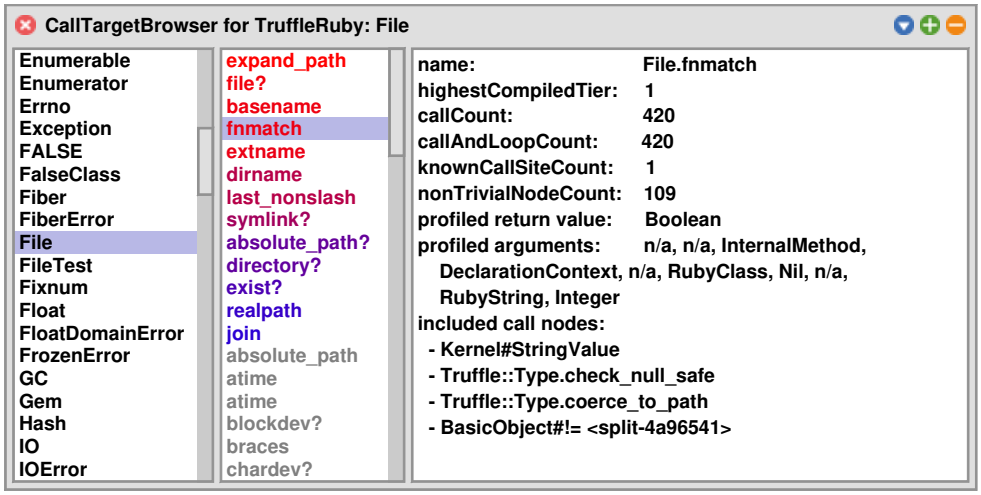


Figure 12.9.: The `CallTargetBrowserRuby` tool allows users to explore `CallTarget` objects from TruffleRuby at run-time similar to how `CallTargetBrowser` allows it for TruffleSqueak.

this case, detailed information on the call target is displayed in the panel on the right, in the same way this information is displayed in the code pane of the `CallTargetBrowser`. This time, however, it is necessary to understand the `Frame` layout used by TruffleRuby to understand profiled arguments. Although the `callCount` is low compared with `Behavior>>includesSelector:` from the previous example, the call target has already been compiled in tier one and includes four other call nodes, one of which was split. The additional call nodes reveal an implementation detail of `fnmatch()`: While CRuby implements `fnmatch()` in C, TruffleRuby has implemented this function in pure Ruby. If this function was implemented on the level of the language implementation, there would have been no call nodes to inline. Furthermore, the example from Figure 12.9 also shows that a major part of the `File` class has not been used yet. Only 13 methods have a call target, the rest is grayed out. `CallTargetBrowserRuby` also supports periodic updates. If, for example, code runs that exercises more of the `File` class, the list of methods would be refreshed allowing developers to observe how Truffle and Graal profile more of the class. Code coverage, on the other hand, is not supported because the required information cannot be derived from TruffleRuby ASTs, which always contain all syntactic nodes on the first execution of a call target.

Selected Implementation Detail TruffleSqueak's VM introspection capabilities make it possible to build tools such as the `GraalInfoMorph` or browsers for call target information. These capabilities are based on GraalVM's support for interoperability with the host Java and on TruffleSqueak's `JavaObjectWrapper`,

which provides unrestricted access to VM-level objects. As a result, language-specific or compiler-specific tools can be built at run-time and do not require any modification on the side of language implementations or the compiler. Furthermore, the `ToolBuilder` framework from Squeak/Smalltalk and the Smalltalk programming experience make the creation of tools very productive. The `CallTargetBrowserRuby` tool is a good example of this: Although it is a self-contained tool, not a subclass of an existing one, it is written in 43 methods with a total of only 193 SLOC. Moreover, only three of these methods are specific to the Ruby language, and one to TruffleRuby:

1. The `moduleAndClassList` method returns a list of all Ruby module and class names of the top scope.
2. The `methodList:` method returns a list of method names for a given Ruby module or class name.
3. The `methodFor:of:` method returns a Ruby `Method` object for a given method name and a module or class name.
4. The `callTargetFor:of:` method uses `methodFor:of:` to retrieve a Ruby `Method` object and returns the corresponding `CallTarget` object from TruffleRuby via TruffleSqueak's `vmObject` infrastructure.

The rest of the tool can be re-used to provide a similar browser for call targets of other GraalVM languages. With only five method overrides and 25 additional SLOC, we created a `CallTargetBrowserPython` subclass for GraalPython to demonstrate this (see Figure C.3).

Insights and Lessons Learned This project demonstrates that TruffleSqueak is a useful exploratory tool-building platform not only for application and tool developers but also for language and runtime developers.

For TruffleSqueak, we needed to implement a Truffle language and both the `GraalInfoMorph` and the `CallTargetBrowser` have proven to be helpful tools to understand how the Graal compiler optimizes a self-sustaining programming system such as Squeak/Smalltalk. The tools can also be used for teaching purposes and to explain the inner works of the compiler. With the `GraalInfoMorph`, it is possible to monitor Graal's compilation activity from within Squeak/Smalltalk to get an understanding of how much work it does over time but also to identify compilation issues. The tool helped us not only to identify regressions in TruffleSqueak's language implementation but also problems and bugs in the Graal compiler. After upgrading to a new release of GraalVM, for example, we noticed that the compilation queue grew excessively in size, which helped to uncover a bug in Graal's splitting heuristic. The large queue sizes we could observe when running polyglot notebooks are another example and were an additional reason for the GraalVM team to rework

Truffle's queuing algorithm. Based on the simple feedback on compilation queue sizes, we further used other tools of TruffleSqueak to examine specific aspects of the compiler in more detail. One example of this is the polyglot notebook in Figure C.2 that we used to analyze snapshots of the compilation queue.

The `CallTargetBrowser`, on the other hand, helped us to understand how Smalltalk methods are optimized by the Graal compiler in more detail. The `highestCompiledTier` value, for example, is a simple indicator for how much effort, if any, Graal has put into optimizing a specific method. Another interesting piece of information that is easily accessible through the tool is whether a primitive method has a call target or not. Since TruffleSqueak tries to eagerly evaluate primitives, a primitive method with a call target reveals that the corresponding VM primitive has failed and as a result, a call target for the method's fallback code was created. Oftentimes, this suggests that something is wrong with the implementation of the primitive, for example, that a Truffle specialization for specific arguments is missing.

Moreover, the fact that TruffleSqueak is a self-sustaining programming system can have side effects when it comes to the compiler and other VM internals. This means that tools built in TruffleSqueak can have a larger influence on the runtime system compared with remote tools. As the `GraalInfoMorph` example shown in Figure 12.6 has demonstrated, tools for inspecting certain aspects of the runtime system can directly influence them and therefore falsify the data they show. The `RLivePlotMorph` visualizing compilation queue sizes over time, for example, can add new compilation tasks to the queue. Similarly, the `CallTargetBrowser` may exercise certain parts of the system more than usual and therefore presents a somewhat biased view over the actual state of call targets present in the system.

Furthermore, the `CallTargetBrowser` allowed us to explore the potential of call target information for application developers: The information on inlined call targets, for example, can be used to allow developers to browse actual implementors of a specific method, not just all implementors found in Squeak/Smalltalk. Similarly, refactoring tools could be improved to allow the renaming of actual implementors. Instead of the `knownCallSiteCount`, call targets also included detailed information on known call sites in an earlier version of Graal. Similar to inline call targets, we could use this information to allow developers to browse actual senders and not just all senders found throughout the system. Moreover, the type information on profiled arguments and return values could also be used for other development features such as code completion. For comparison, the Live Typing project [215] required non-trivial changes to the OpenSmalltalkVM to provide the same functionalities. In TruffleSqueak, on the other hand, all of this comes for free because the

Graal compiler already collects a lot of profiling information that can easily be accessed, for example, through call targets.

Lastly, we learned that the use of command-line flags often influences the implementation of compiler optimizations and options: Since command-line flags cannot be changed at run-time, their input values are often assumed to be immutable. This means that, although it is possible to inspect values, for example, that configure compiler heuristics, they are often not designed to be changed while the compiler is running. Revising compiler optimizations and options so that they can handle configuration changes at run-time, on the other hand, would allow further exploration of compiler internals through systems such as TruffleSqueak. The same applies to Truffle-based language implementations. With TruffleMATE [24], Chari *et al.* have shown that it is possible to build a fully reflective VM in Truffle. Expanding this idea to the host VM and combining it with our approach could allow developers to evolve polyglot VMs and language implementations at run-time, but is left for future work.

12.5. Extending Squeak/Smalltalk With a Polyglot Drawing Engine

> **Research Question** What are the benefits and drawbacks of using polyglot programming with GraalVM to build a new drawing engine for Squeak/Smalltalk?

Problem Statement Squeak/Smalltalk uses the BitBlt and Balloon VM plugins for rendering its programming system. As discussed in Section 8.1, we ported these two plugins from C to Java to improve TruffleSqueak's UI performance. BitBlt, however, was originally designed for the Xerox Alto in the 1970s [71]. Although the BitBlt implementation used by Squeak/Smalltalk has evolved over the years, it conceptually still works the same. The way graphics work today, on the other hand, has changed significantly. Both rendering plugins, for example, run on the CPU and consequently do not benefit from hardware acceleration provided by modern GPUs.

The outdated architectures of BitBlt and Balloon, however, have become more and more noticeable with the increase in display resolutions. As a result, the Smalltalk community has explored different approaches (e.g., [40, 98, 186]) to build new drawing engines that, for example, support vector rendering and are based on more recent graphics technologies such as OpenGL [200], Cairo [219], or Skia [55]. These approaches often use C-based foreign function interfaces for integrating low-level graphics libraries. In this case study, we

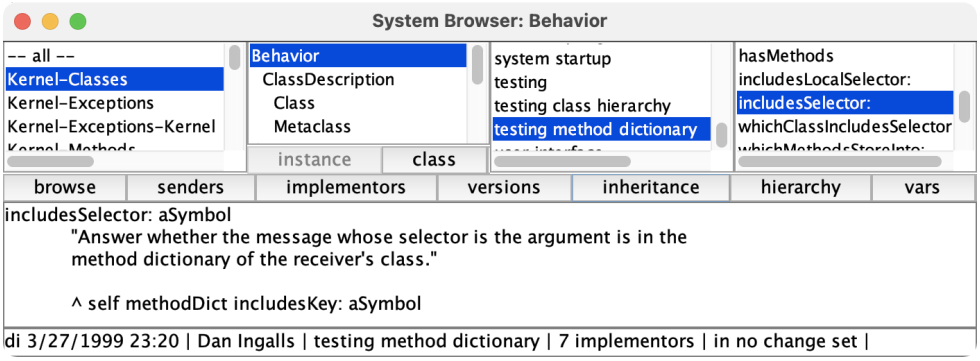


Figure 12.10.: The `Browser` tool from Squeak/Smalltalk built in Java Swing through our `JavaSwingToolBuilder`.

explored how a new drawing engine for Squeak/Smalltalk could be built through polyglot programming with GraalVM.

Project Outcome Many tools in Squeak/Smalltalk are implemented using `ToolBuilder`. The `ToolBuilder` framework follows the builder pattern [50, pp. 97–106] and constructs tools from appropriate specifications provided by tool developers. More specifically, `MorphicToolBuilder` constructs tools using the Morphic UI framework.

In the first exploration phase, we created a `JavaSwingToolBuilder` that can construct tools with Java Swing for the same tool specifications. Apart from Smalltalk, Java is the only other language well-supported by GraalVM that provides UI frameworks as part of its standard library. Since Espresso, the Java language implemented in Truffle, did not support Swing at that time, we used the host Java instead. Since interoperability with the host Java works through GraalVM's interoperability protocol the same way it works for guest languages, this gave us a realistic impression of building a polyglot drawing engine. Multi-language debugging and dynamic code evaluation are, however, limited to guest languages. On the other hand, host Java does not suffer from warmup issues that were present in Espresso at that time.

Figure 12.10 shows a screenshot of Squeak/Smalltalk's `Browser` tool built with the `JavaSwingToolBuilder`. Internally, this builder implements different, abstract specifications for UI components using appropriate Swing components. For a `PluggableButtonSpec`, for example, `JavaSwingToolBuilder` creates a Swing `JButton`, a `PluggableTextSpec` creates a `JTextArea`, and so on. As a result, the UI is entirely rendered by Swing widgets with proper support for high-resolution displays. User events, however, are forwarded to and still handled by the Squeak/Smalltalk model. As Figure 12.10 demon-

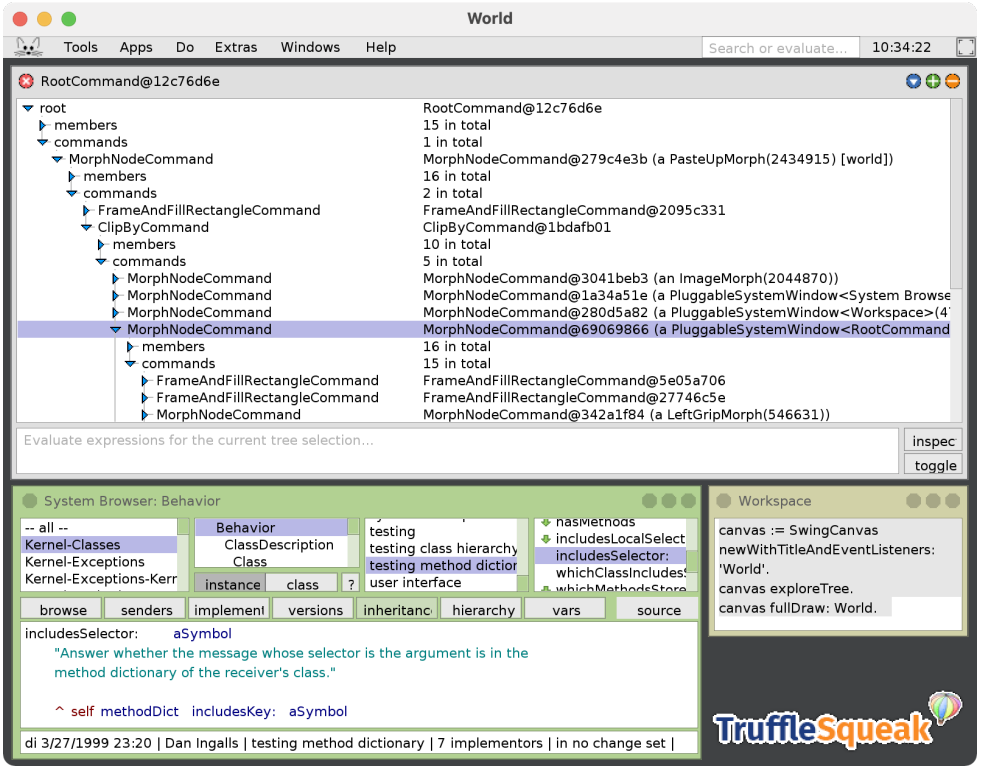


Figure 12.11.: The TruffleSqueak `World` drawn by Java Swing through a `SwingCanvas`.

strates, tools can further be built in individual native windows, which enables a multi-window experience in Squeak/Smalltalk.

This prototype only required 59 SLOC of Java code and thus was mostly built in Squeak/Smalltalk. However, the `JavaSwingToolBuilder` approach has a major limitation. Some tools, such as the `PreferenceBrowser`, are not implemented in the `ToolBuilder` framework and can therefore not be built by `JavaSwingToolBuilder`. The same is true for some tools that work around the framework to implement certain features directly in Morphic. TruffleSqueak's `PolyglotNotebook`, for example, allows code cells to be added, removed, and reordered, something that `ToolBuilder` does not support.

In the second phase of exploration, we thus focused on the Morphic framework and worked on a new `Canvas` implementation based on Java Swing. This way, all tools and applications written in Morphic can be supported.

The result can be seen in Figure 12.11: Since the Squeak/Smalltalk `World` object uses Morphic, the entire programming system can be drawn with our `SwingCanvas` using Java Swing. The screenshot in Figure 12.11 shows a TruffleSqueak environment with three open tools. The window on the top is a `DrawTreeExplorer`, a tool we built during the development of `SwingCanvas`.

With this tool, we can explore two aspects of our `SwingCanvas` implementation: The tree of draw commands that our `SwingCanvas` has recorded following the command pattern [50, pp. 233–242] and the tree of Swing components that were created for these draw commands. The "toggle" button underneath the "inspect" can be used to switch between these two views allowing us to compare the two trees, which helps to identify transformation issues among others. In this case, the `RootCommand` opened in the tool contains one `MorphNodeCommand`. As suggested by their display string, all these objects are Java objects. The one `MorphNodeCommand`, in particular, is responsible for drawing the `PasteUpMorph` that represents the `World` object in Squeak/Smalltalk. This command, in turn, contains two sub-commands: A command for drawing a filled rectangle and a `ClipByCommand`. The latter again has sub-commands, which are `MorphNodeCommands`. The one selected is responsible for drawing this very `DrawTreeExplorer` opened in the screenshot. Its first two sub-commands, for example, draw the outermost border and the gray background rectangle of the tool. For comparison, the screenshot also shows a `Browser` tool opened on the same method that is also shown in Figure 12.10. The workspace above the TruffleSqueak logo contains the *doIt* that created a new `SwingCanvas`, opened the upper `DrawTreeExplorer`, and instructed the canvas to draw the `World` object.

The screenshot also shows some limitations of the implementation, mostly due to time constraints: At the time of writing, `SwingCanvas` can only draw Smalltalk `Forms` of 32-bit depths and does not properly support transparent `Forms` yet. Although the font used by AWT for rendering matches the one used in Squeak/Smalltalk, the positioning of text is not always aligned, and different font widths are not yet considered. This is due to subtle differences in how Squeak/Smalltalk and Swing layout texts.

Selected Implementation Detail Swing is based on Java AWT, which uses OS-level threads for rendering. In Squeak/Smalltalk, on the other hand, rendering happens as part of the main thread. This architectural difference has consequences in both the `JavaSwingToolBuilder` and the `SwingCanvas` implementations. Mouse and keyboard events, for example, are handled in AWT threads. From there, however, they cannot invoke the corresponding event handler in Smalltalk as it does not support OS-level multi-threading. To compensate for that, `JavaSwingToolBuilder` uses a dedicated AWT `ActionListener` that records events in a Java queue, which is then processed through a `JavaEventSensor` in Smalltalk and on the main thread.

A simplified implementation of this `ActionListener` is depicted in Listing 12.6: To instantiate such a listener, an `eventCallback` of the type `Value` is passed in. The `Value` type is provided by the GraalVM SDK [138], used to repre-

Listing 12.6: Simplified implementation of an AWT `ActionListener` that adds event callbacks to a thread-safe `Queue`.

```
public class ToolBuilderActionListener implements java.awt.event.ActionListener {
    private static Queue<Value> EVENT_QUEUE = new ConcurrentLinkedQueue<>();

    private Value eventCallback;

    public ToolBuilderActionListener(final Value eventCallback) {
        this.eventCallback = eventCallback;
    }

    @Override
    public void actionPerformed(final ActionEvent e) {
        EVENT_QUEUE.add(eventCallback);
    }
}
```

Listing 12.7: Simplified implementation of the `JavaEventSensor>>processEvents` method that polls the Java `Queue` for event callbacks and activates them in a new Smalltalk process.

```
processEvents
    | eventCallback |
    [ eventCallback := self eventQueue poll ] whileNotNil: [ eventCallback fork ]
```

sent guest language objects, and facilitates communication through GraalVM's interoperability protocol. The `JavaSwingToolBuilder` creates these listener objects passing in an appropriate Smalltalk `BlockClosure` as `eventCallback` and adds them to Swing UI components such as `JButtons`. When such a button is clicked, for example, the `ToolBuilderActionListener` adds the corresponding `BlockClosure` to the `EVENT_QUEUE`.

A Smalltalk-level `JavaEventSensor` process, inspired by Squeak/Smalltalk's `EventSensor`, frequently invokes its `processEvents` method shown in Listing 12.7. This method has access to the Java queue, polls for event callbacks represented by Smalltalk `BlockClosures`, and activates them through `fork` in a new Smalltalk-level process. This way, callbacks run independently from the `JavaEventSensor` process. A closure handling a button click event, for example, invokes the corresponding method that was specified as part of the `PluggableButtonSpec` that was part of the tool's original specification.

Note that for this prototype, this infrastructure was sufficient. To distinguish between different types of events, the `ActionEvent` objects could further be made available to Smalltalk event callbacks. For this, both the event callback and the corresponding `ActionEvent` object could be added to the event queue. In the `JavaEventSensor`, event callbacks can then be activated supplying the `ActionEvent` object as an argument. Moreover and in addition to its `EventSensor`, Squeak/Smalltalk also processes user events on every frame. This could also be done for events from Java to further improve responsiveness.

Insights and Lessons Learned This project has demonstrated that not only TruffleSqueak's tools but also a large part of the programming system can be turned into non-trivial polyglot applications. From these efforts, we gained further insights, for example, into what it means to use frameworks through polyglot programming as opposed to libraries. Although Swing and AWT are often referred to as toolkits, some parts of them make use of inversion of control [50, p. 27], a characteristic that typically distinguishes frameworks from libraries. As illustrated as part of the selected implementation detail, for example, event handling is implemented in AWT that way.

An insight gained through this particular example is that polyglot programming across languages that support threading in different ways, if at all, can be challenging. Appropriate implementation strategies are needed to make this possible. Another example of this is `SwingCavas`' use of the command pattern to record drawing operations from Morphic in Java objects within the main thread, which are then accessed by AWT's rendering thread.

Furthermore, we encountered situations where objects from one language need to be copied into corresponding objects from the other language, something that polyglot programming with polyglot VM tries to avoid. Examples are Smalltalk `Rectangle` objects that we needed to copy to AWT `Rectangle` objects or `Point` and `Dimension` objects from AWT from which we created Smalltalk `Point` objects. Some reasons for this are interface mismatches, which are not uncommon during polyglot programming. The values for `x` and `y` of a Smalltalk `Point`, for example, can be set through the `setX:setY:` method. AWT's `Point` class, on the other hand, provides multiple, overloaded `setLocation()` methods for this, while corresponding methods in its `Dimension` class are called `setSize()`. But even if there were no interface mismatches, we would have needed to copy some Smalltalk objects or introduce appropriate adapters, if possible, due to Java's type system. Target type mappings [142], a feature supported by GraalVM's SDK [138], helps to automate the conversion of guest language objects to objects of the host language. Nonetheless, these mappings do not avoid object copies or allocations of appropriate adapters. In Section 14.4, the topic of interface and type mismatches is discussed further.

Moreover, we found that the overall programming experience building `JavaSwingToolBuilder` and `SwingCanvas` was better compared with CFFI-based approaches. The main reason for this is that we were able to stay on the level of two high-level programming languages and could focus on connecting APIs that work on comparable levels of abstraction. Integrating OpenGL and other low-level graphics APIs, on the other hand, usually requires additional knowledge on low-level abstractions such as on the architecture and the mechanics of modern GPUs. Some of these low-level APIs are sometimes even

platform-specific, which can require platform-specific glue code. Our polyglot implementation, on the other hand, is fully portable. Another advantage of working with two high-level and memory-safe languages is better error handling. In case of errors, we were able to use high-level debuggers for Smalltalk and Java, which can be especially helpful when exploring APIs. Once Espresso can be used, we can even step from Smalltalk code into Java code and vice-versa within a single debugger. When working with CFFIs, on the other hand, crashes due to invalid memory accesses or other incorrect uses are not uncommon, especially during exploration. Since the debugger of the high-level language cannot step into a CFFI call, developers have to switch to OS-level debuggers such as gdb to debug crashes in C.

Summary As part of our fifth contribution, we present five case studies that show how TruffleSqueak can be used to approach various research questions on tools for and applications of polyglot programming:

1. By composing existing tools, we can quickly build a Jupyter-inspired polyglot notebook system on top of GraalVM. Based on this, we explore, for example, how variables can be shared across languages automatically to improve usability.
2. We build a code editor that allows us to explore how the different polyglot APIs of GraalVM languages can be integrated into an editor. Code boxes, for example, allow developers to embed code from other languages within our editor and within other boxes while completely automating the use of the polyglot APIs for the user.
3. Our polyglot code finder tool allows developers to search for reusable code written in different languages on StackOverflow and from within other tools such as our notebook or editor. Code from search results can be interactively evaluated before use and even filtered by providing example input and output values.
4. The VM introspection capabilities of TruffleSqueak allow us to build tools that help to understand the dynamic behavior of the Graal compiler. Based on this, we build a tool to monitor the size of Graal's compilation queue while it is running. Another tool connects profiling information collected by the compiler with source code, allowing developers to observe how the compiler optimizes user applications as well as the entire programming system at run-time.
5. We conduct two experiments that make extensive use of polyglot programming to allow Squeak/Smalltalk tools to be rendered through Java UI frameworks. GraalVM and TruffleSqueak allow us to build appropriate infrastructures based entirely on the high-level APIs of these frameworks, create cross-language helper tools, and use high-level debuggers in case of errors.

All of these tools and prototypes are themselves built in a polyglot way. This allowed us to gain practical experiences applying polyglot programming, which we summarized for each study.

13. Case Studies Beyond TruffleSqueak

With TruffleSqueak, we have presented an implementation of our approach for the GraalVM. The case studies from Chapter 12 have further demonstrated how it can be used to explore different tooling ideas in the context of polyglot VMs. This chapter presents two case studies that go beyond TruffleSqueak: The first study demonstrates that our approach can also be applied to a polyglot VM built with RPython. For this, we compose existing interpreters for Python, Smalltalk, and Ruby, and show that the meta-object protocol from Squeak/Smalltalk can be used to provide language interoperability. In the second study, we present and discuss two polyglot notebook systems that are based on insights gained through the `PolyglotNotebook` presented in Section 12.1. This illustrates that insights from TruffleSqueak are not limited to it and can be transferred to other programming systems.

13.1. Applying Our Approach to a Polyglot VM Built With RPython

In Part IV, we have shown how our approach can be applied to create an exploratory tool-building platform for the GraalVM. While GraalVM is among the most advanced VMs designed specifically for polyglot programming, our approach is neither limited to it nor its general architecture. This section gives a brief overview of Squimera, an exploratory tool-building platform for a polyglot VM based on the RPython language implementation framework [165]. Instead of building another language implementation to host a self-sustaining programming system on another polyglot VM, we apply interpreter composition [5] to compose three existing interpreters in RPython to create our own polyglot VM: 1) PyPy [165], a Python interpreter as part of which RPython is maintained, 2) RSqueak/VM [15, 42], another Squeak/Smalltalk VM, and 3) Topaz [202], a Ruby interpreter. While this allows us to re-use existing language implementations, we need to build our own infrastructure to allow the three languages to communicate with each other. In the following, we present Squimera, discuss lessons learned, and compare it with TruffleSqueak and GraalVM.

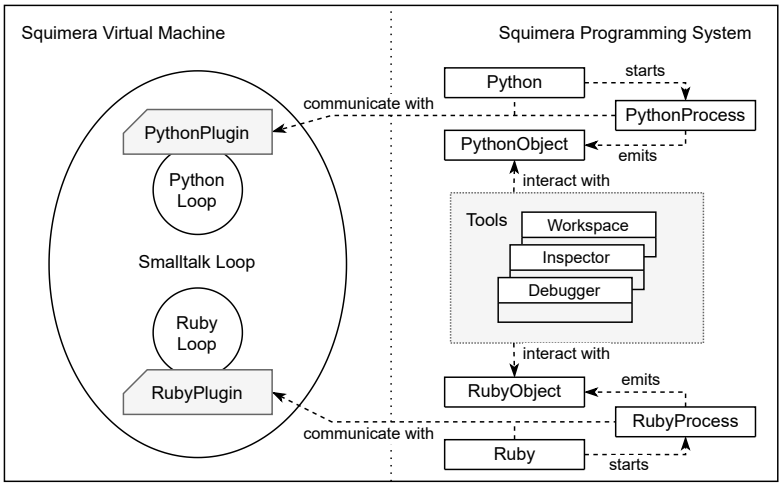


Figure 13.1.: Architectural overview of Squimera's virtual machine and its programming system.

Project Outcome Although Squimera also uses Squeak/Smalltalk as its self-sustaining programming system, it is different from TruffleSqueak in many ways. One example is that its VM contains three bytecode interpreter loops, one for each language. In Truffle, on the other hand, all languages are implemented as AST interpreters. Another important difference is that the languages supported by Squimera do not run on top of Python, the language they are implemented in. All Truffle languages, on the other hand, run on top of Java because GraalVM is based on the JVM. This also means that Squimera needs to deploy its own execution model to interpret different languages. To give the SSPS full control over the execution of code from other languages, Squimera builds on the execution model of Squeak/Smalltalk. Since RPython does not provide any means for language interoperability, Squimera's VM only provides a basic set of primitives to evaluate code in Ruby and Python, to send messages to foreign objects, to interact with stack frames, and to convert foreign objects to Smalltalk. The rest of the infrastructure for languages to interoperate is implemented within Squimera's programming system.

Figure 13.1 provides an overview of Squimera's architecture. It shows the interpreter loops of RSqueak/VM for Smalltalk, of PyPy for Python, and of Topaz for Ruby. Through two VM-level plugins, the Squeak/Smalltalk-based programming system can interact with the interpreters for Python and Ruby. Inside the programming system, there are three different classes for each Python and Ruby: The `Python` and `Ruby` classes provide access to the corresponding language, allowing, for example, the evaluation of code. When code from Python and Ruby is executed, either a `PythonProcess` or `RubyProcess` is created and activates the corresponding interpreter loop from PyPy or Topaz.

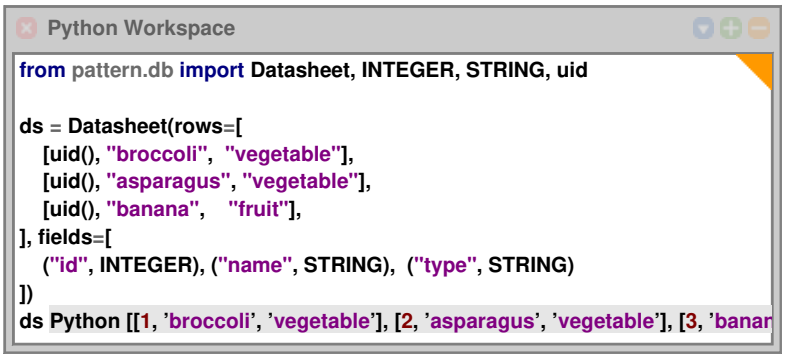


Figure 13.2.: Interactively evaluating Python code in Squimera's polyglot workspace.

We have patched both interpreters so that they periodically yield back to the Smalltalk interpreter loop. This allows the Squeak/Smalltalk scheduler to switch to another Smalltalk-level process, which can be another language process, its UI process, or some other process from Squeak/Smalltalk. This way, our system remains in control over the execution of foreign code.

Furthermore, foreign objects are represented by either the `PythonObject` class or the `RubyObject` class. Similar to TruffleSqueak's `ForeignObject`, these two classes override the reflective methods used by the exploratory tools of Squeak/Smalltalk to access information on the structures and interfaces of foreign objects. This means that Squimera uses the meta-object protocol from Squeak/Smalltalk as its interoperability protocol. The implementation is entirely built on the VM primitives to evaluate code, to send messages to foreign objects, and to convert foreign objects to Smalltalk.

In addition and similar to TruffleSqueak, Squimera provides adaptations of different tools of Squeak/Smalltalk including its exploratory programming tools as well as an adaption of the debugger:

Figure 13.2 shows a screenshot of Squimera's polyglot workspace tool. Its base language is set to Python, which allows the interactive evaluation of Python code. In this case, the user interacts with `pattern`, a web mining module for Python, to create a new `Datasheet` object. The *doIt* in the last line gives a glimpse of what the resulting object looks like. The language of foreign objects, Python in this case, is used as a prefix in the display string to make it clear from which language an object is from. Similar to TruffleSqueak's `PolyglotWorkspace`, the workspace also supports syntax highlighting for Python and Ruby code. Instead of the Rouge library for Ruby, it uses Pygments, a similar syntax highlighter written in Python.

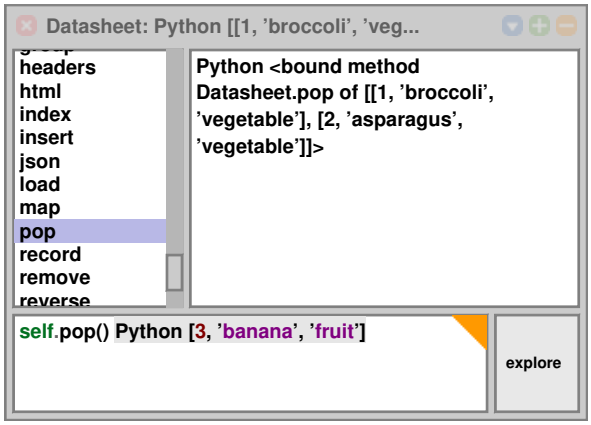


Figure 13.3.: A polyglot inspector opened on the `Datasheet` object instantiated in Figure 13.2.

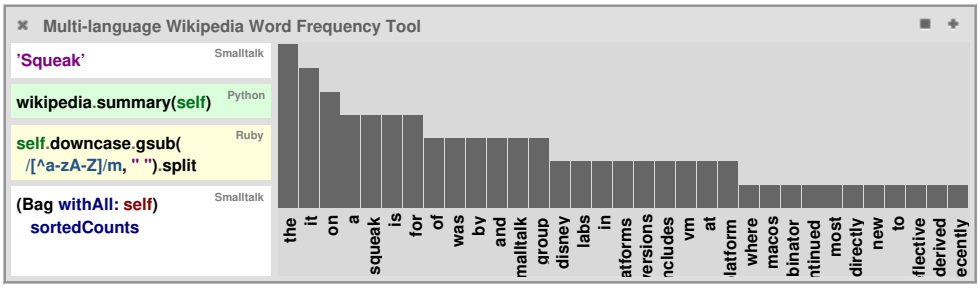


Figure 13.4.: A tool for measuring the word frequency of summaries from Wikipedia built with Vivide and written in Smalltalk, Python, and Ruby.

Squimera's polyglot inspector is shown in Figure 13.3. The inspected object is the `Datasheet` object created in the workspace from Figure 13.2. The left pane lists the elements as well as the interface of the object under inspection. With that information, the user can find out that the object implements a `pop` method. On the right, the inspector reveals that `pop` is indeed a method bound to the `Datasheet` object. Using the embedded workspace, which also supports syntax highlighting based on Pygments, it is possible to evaluate code in the context of the inspected object using the language of its origin. In this case, the user has called the `pop` method on the object, which returned the last element from the `Datasheet` object. The live feedback mechanism of the original inspector tool from Squeak/Smalltalk can also be re-used. This is, for example, the reason that the display string for the bound method of the `Datasheet` object shown in the right pane has already been updated and no longer shows the third element because it was removed from the object.

Furthermore, we can also build polyglot applications and tools in Squimera. Figure 13.4, for example, shows a notebook-like tool for measuring the word fre-

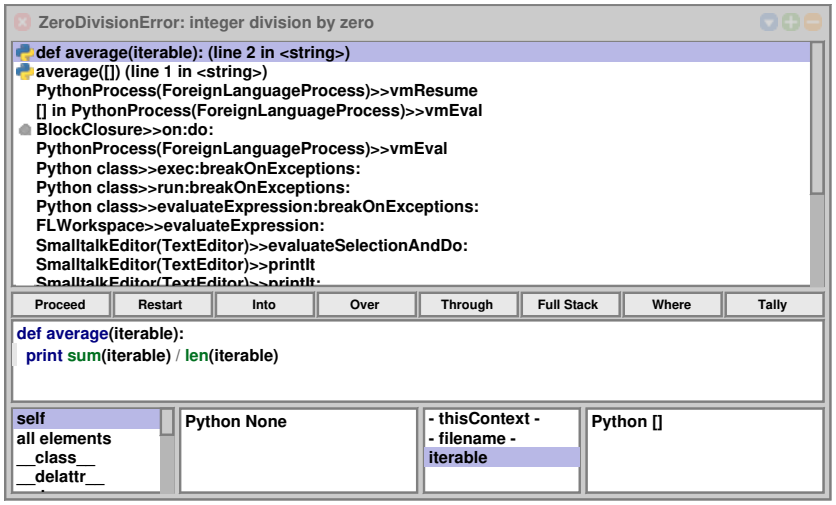


Figure 13.5.: Debugging a `ZeroDivisionError` from Python in Squimera.

quency of Wikipedia summaries built with Vivide [196], a Squeak/Smalltalk framework for data-driven tool construction. The tool consists of four connected, workspace-like boxes. Each of them can be set to a different language. The first box contains the search term as a Smalltalk string, which is passed to the next box. In the Python box, the result from the previous box is bound to `self`. More importantly, the Python box passes the Smalltalk string into the `summary` method of the `wikipedia` library, which queries the Wikipedia API and returns the summary text found for the given search term. The next box uses Ruby to extract a list of lowercase words from the Python string. Afterward, a Smalltalk `Bag` is used to aggregate the list of words before the result is visualized with a histogram on the right.

Since Smalltalk is in control of the execution of foreign code, we can extend the debugger in Squimera with support for foreign languages. Figure 13.5 shows a screenshot of the debugger opened on a `ZeroDivisionError` from Python. The list of stack frames reveals that this error occurred in an `average` method, which was called as part of a *printIt* triggered in Squimera's workspace. A little icon indicates which stack frames are from Python. Furthermore, the code pane shows the source of the `average` method with Pygments-provided syntax highlighting. Since the method is defined in Python's top scope, `self` is bound to Python's `None` object. More importantly, the scope inspector on the lower right shows that `iterable` is an empty Python `list`, which has led to the `ZeroDivisionError`. Squimera patches PyPy and Topaz not only so that they periodically yield back to the interpreter from RSqueak/VM but also with experimental support for hot-code reloading. This means that within this debugger session, the code of the shown stack frame can be changed. When a change is applied, the debugger will restart the current stack frame

with the updated code. This way, it is possible to recover from Python and Ruby errors, similar to how it is possible to recover from errors in Smalltalk.

Lessons Learned and Comparison to TruffleSqueak Squimera demonstrates that our approach can also be applied to a polyglot VM based on RPython. Both Truffle and RPython provide appropriate building blocks for the construction of interpreters and make implementing languages more productive by allowing language developers to use high-level languages. The two frameworks also come with noteworthy differences that have a direct impact on our two implementations.

Truffle languages and thus TruffleSqueak run on top of Java, which predetermines the default execution model and allows VM introspection through interoperability with the host language. Although RPython allows language developers to write interpreters in a subset of Python, it uses a C backend to translate implementations to C, which is then compiled into a binary. This means that RPython-based interpreters and consequently their compositions including Squimera do not use Python as a host language. This means that inspecting VM internals cannot easily be done through interoperability with the host language. Due to time constraints, we did not work on an infrastructure that makes, for example, the RPython JIT compiler or garbage collector accessible from within Squimera. For the same reason, we also did not fully implement the ability to call Smalltalk methods from Python or Ruby.

The lack of a protocol for language interoperability in RPython caused additional work as we had to manually connect both languages with Smalltalk and Squimera's programming system. This as well as the fact that PyPy is the only production-level language implementation in RPython are further reasons why we did not incorporate additional languages such as Racket [8] or Prolog [6]. On the other hand, the additional work of mapping languages to the Smalltalk meta-object protocol, which we used as our interoperability protocol, enabled the construction of language-agnostic and polyglot-aware tools.

Moreover, Squimera has more control over the execution model. As we were able to demonstrate, this makes it possible to re-use the debugger from Squeak/Smalltalk for Python and Ruby, without having to run these languages in separate threads or processes. The use of Smalltalk processes also allows interruption of long-running evaluation requests, such as when launching server applications. The additional control over the execution of foreign languages also allowed us to explore approaches to enable hot-code reloading in RPython-based language implementations. The fact that Python, Ruby, and many other languages follow the termination model for exception handling [21], however, makes it hard to detect unhandled exceptions at the

time they are raised and before the stack is unwound. The resumption model as used in Smalltalk, on the other hand, does not have this problem because control flow can return to the raise point, for example, through the use of continuations [161] or similar mechanisms.

Even though Squimera can be used as a platform for exploratory programming and tool-building in the context of an RPython-based polyglot VM, it is not as extensive as TruffleSqueak. For one, the protocol for language interoperability is limited to the capabilities of the basic `Object` protocol of Squeak/Smalltalk. Truffle's protocol, on the other hand, supports several interoperability traits and types. It also does not properly allow the evaluation of Smalltalk code from Python or Ruby. Furthermore, Squimera does not provide extensive VM introspection capabilities and is, therefore, less useful to language and runtime developers. While it is possible to expose VM internals to guest languages, additional work is required. Since Truffle and GraalVM provide access to the host language through the interoperability protocol, VM introspection comes almost for free in TruffleSqueak. Only the `JavaObjectWrapper` infrastructure and some configuration of the Java module system were needed to provide unrestricted access to GraalVM internals. Nonetheless, we believe that Squimera demonstrates that our approach is not limited to GraalVM and that it can also be applied to other polyglot virtual machines.

13.2. Bringing Polyglot Notebooks to Jupyter and VS Code

In TruffleSqueak, we were able to compose existing exploratory tools to rapidly build a simple notebook system, as illustrated in Section 12.1. This system then allowed us to explore how GraalVM languages can be integrated into notebooks and how users of such polyglot notebooks can be further supported with features such as automatic variable sharing across languages and an object inspector for these shared variables. While TruffleSqueak's `PolyglotNotebook` supports the Jupyter notebook format, it does not implement the client-server-based architecture that is commonly used by notebook systems such as Jupyter or Google Colab. Since it is built and runs inside a self-sustaining programming system, it can, however, directly interact with the underlying runtime system, for example, to execute code cells written in different languages.

To evaluate the extent to which the insights we gained when exploring polyglot notebooks in TruffleSqueak are transferable to other systems, we conduct two experiments: In the first experiment, we want to find out what it takes to bring features of our polyglot notebook system to Jupyter. The

second experiment explores a new implementation strategy for a polyglot notebook system based on the Language Server Protocol and for the notebook UI provided by VS Code. These two experiments further allow us to get an impression of how productive exploration in TruffleSqueak is compared with other programming systems. In the following, we present the results of both experiments and compare them with our experiences building the `PolyglotNotebook` in TruffleSqueak.

Polyglot Kernel and Extension for Jupyter To allow polyglot programming within Jupyter notebooks, we need to implement a notebook kernel based on GraalVM. But instead of creating a new kernel from scratch, we choose to build on IJavascript [162], a JavaScript kernel implemented on top of Node.js. On the one hand, this keeps the implementation effort low because IJavascript already implements the protocol of notebook kernels. On the other hand, a Node.js-based kernel also allows the reuse of Node.js packages, something that is not supported by GraalVM's polyglot API at the time of writing.

The main work required to turn IJavascript into our IPolyglot kernel [182] is to redefine how the kernel evaluates code. Instead of evaluating code through Node.js' `vm.runInThisContext` API, we patch the NEL module that IJavascript uses so that it can call out to other languages through Graal.js' polyglot API. Since Graal.js neither provides access to the polyglot bindings object nor the ability to pass in a local scope when evaluating code in other languages, we can only implement automatic variable sharing in a language-specific way. This involves language-specific methods to identify language globals as well as to generate code for exporting and important variables across GraalVM languages. Furthermore, the system uses a simple JavaScript object instead of the polyglot bindings object as a key-value store for shared variables. In addition to that, we also build on the Variable Inspector extension [75] for Jupyter to create an inspector for automatically shared variables, similar to the explorer in the sidebar of TruffleSqueak's `PolyglotNotebook` tool.

Figure 13.6 shows a screenshot of a polyglot notebook in our extended version of Jupyter, inspired by the notebook shown in Figure 12.1. The top right corner of the navigation bar reveals that the Jupyter notebook UI is connected to our IPolyglot kernel, which runs on top of Graal.js and hence GraalVM. Jupyter's UI, however, is based on the assumption that only one language is used in a notebook. Therefore, it does not provide any means to select a specific language per code cell. Instead, we introduce a new `%polyglot` magic command that can be used in the first line of a code cell to specify the language. This way, no additional work is needed to persist language selections in Jupyter notebook files. Apart from that, the three code cells written in Ruby, Python, and R do conceptually the same as the notebook

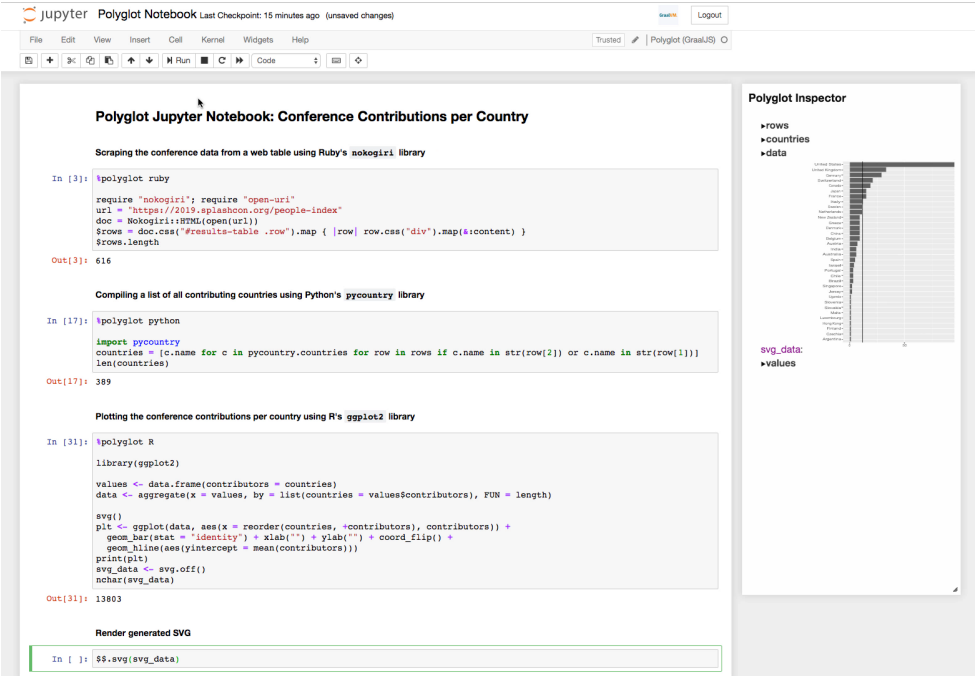


Figure 13.6.: A polyglot notebook for analyzing conference contributions per country in Jupyter, inspired by the notebook from Figure 12.1, and executed by IPolyglot.

in Figure 12.1: The Ruby code cell downloads a list of all contributors from a conference website and uses `nokogiri` to extract a Ruby array containing affiliations and countries from the HTML file. In Python, the `pycountry` library is used to find country names within the array. The third code cell aggregates the Python list of country names in R and generates an SVG string for a plot with the `ggplot2` R package. The last code cell runs a command from the original IJavascript, which instructs the notebook UI to render a string as an SVG within the corresponding output cell. While interacting with the notebook and IPolyglot, the polyglot inspector on the right provides an up-to-date overview of the variables that are automatically shared. The inspector has detected that the `svg_data` variable holds a valid SVG string and thus renders the SVG inline. All other variables can be inspected in a tree-like manner similar to how the object explorer tool works in Squeak/Smalltalk.

Polyglot Notebooks via the LSP in VS Code Since IPolyglot is a conventional notebook kernel, it can also be used from within other notebook UIs such as Google Colab or VS Code notebooks. As the previous experiment has shown, official GraalVM guest languages such as Graal.js neither allow language-agnostic nor polyglot-aware tool-building as supported by

TruffleSqueak. Instead of repeating the previous experiment based on a Java-based kernel implemented in the GraalVM SDK [138], we want to explore an entirely new approach for the execution of notebooks: via the Language Server Protocol [111]. The LSP is designed to decouple IDEs and tools from languages. In addition to language features, such as for code completion or go-to definition, the protocol supports custom commands that a client can trigger within a language server. Similarly, language servers can send custom notification messages to LSP clients. GraalVM already provides a language-agnostic LSP server as well as an extension for VS Code that takes care of connecting VS Code with GraalVM's language server. Therefore, we can extend both components to create another polyglot notebook system. On the server side, we add new custom LSP commands for creating notebook sessions, for executing code of code cells for a particular language in a specific notebook session, and for interrupting the execution of code with a session. Execution results, the output of stdout and stderr, error messages, and other data for notebooks are transmitted via appropriate notification messages. For the client, we have to create a new VS Code extension to connect its notebook UI with the extended GraalVM language server based on the connection provided by GraalVM's extension for VS Code. This also allows us to instruct the UI to display the language of each code cell and to provide a dialog that can be opened to select a specific language.

A screenshot of a polyglot notebook opened in our LSP-based polyglot notebook system is shown in Figure 13.7. Again, the example is inspired by the notebook from Figure 12.1. The language of each code cell is displayed in the bottom right corner and can be changed with two clicks. In this prototype, however, the export-import functionality of the polyglot APIs must be used explicitly because automatic variable sharing is not yet supported. Since the language server extension is completely language-agnostic, automatic variable sharing can, however, be implemented without having to rely on language specifics. The GraalVM SDK further provides access to the list of available languages, which makes it possible to build polyglot-aware features such as the dialog for selecting languages.

Observations and Findings The two experiments on GraalVM-based notebook systems provide good grounds for discussing some advantages and limitations of our approach.

For creating the IPolyglot kernel, only a small patch [181] with 143 SLOC for IJavascript's NEL module was needed to route evaluation requests through Graal.js' polyglot API. While this kept the effort of implementing a kernel low, language-agnostic tool-building is limited to Graal.js' polyglot API. Additional features such as automatic variable sharing cannot be implemented

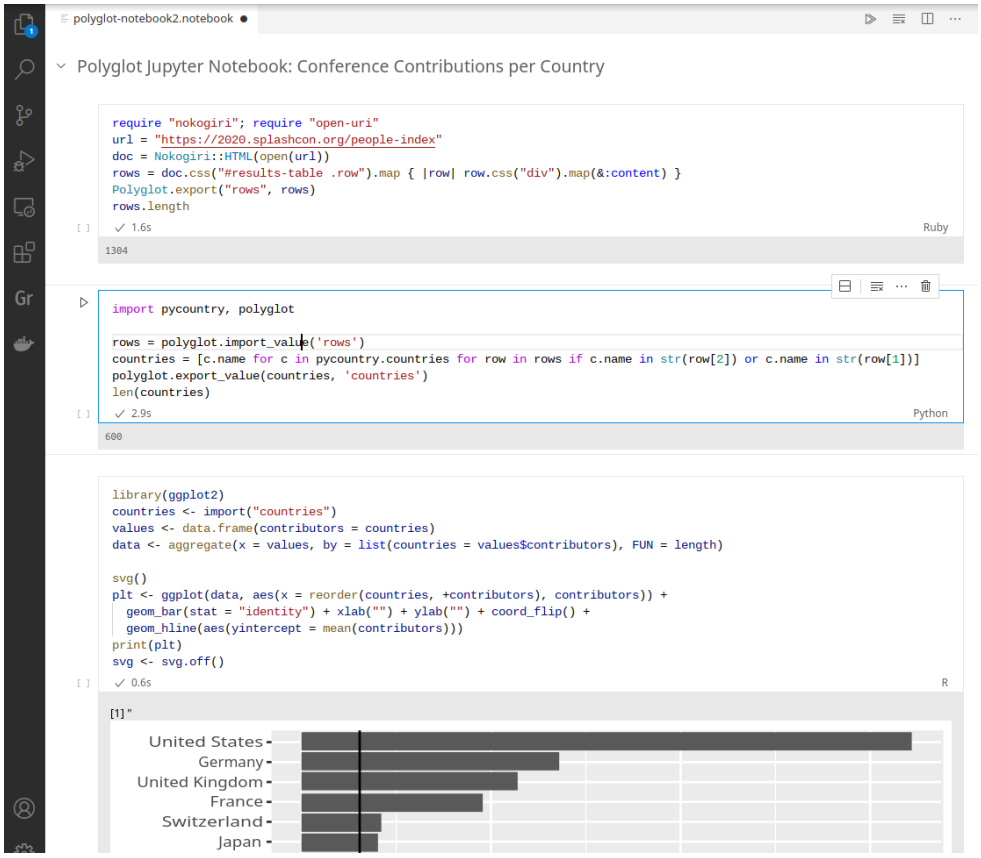


Figure 13.7.: A polyglot notebook for analyzing conference contributions per country within VS Code, inspired by the notebook from Figure 12.1, and executed by the GraalVM language server through the LSP.

in a language-agnostic way because the API neither provides access to the interoperability protocol or globals of languages nor does it allow users to pass in a local scope when evaluating code in other languages. Similarly, there is no way to make the notebook system aware of the polyglot environment. A list of available languages can, for example, not be retrieved from within Graal.js.

These limitations do not apply to our LSP-based notebook system. Instead, we needed to create two independent extensions, one for GraalVM's language server and one for VS Code. The implementation of each of these components is, however, larger than the entire implementation of the `PolyglotNotebook` in TruffleSqueak: The VS Code extension is written in 804 SLOC of TypeScript, the extension for GraalVM's language server in 788 SLOC of Java, and the `PolyglotNotebook` in 725 SLOC of Smalltalk. Since the two extensions build on non-trivial components, the overall complexity of the two notebook

systems developed in the two experiments is also much higher compared with the system we built in TruffleSqueak. We found that this complexity as well as the client-server architecture used in both systems make exploration of user-facing features harder compared with TruffleSqueak. While we could quickly build a first prototype in TruffleSqueak that was tailored to our exploration needs, a lot of boilerplate code was needed to build the other two systems. More importantly, the separation of notebook UI and kernel hinders debugging and can increase feedback loops significantly: A UI change typically requires the UI to be reloaded entirely. To see the effect of a kernel change, kernels usually need to be restarted. And two different debuggers are needed to debug each of these components. In contrast, TruffleSqueak provides the Smalltalk programming experience, which allowed us to build and evolve the `PolyglotNotebook` at run-time and to resolve errors within the Smalltalk debugger while it is running.

Furthermore, extension mechanisms often impose limitations on what third parties are allowed to do. We especially noticed this when we built the extension for VS Code: Although VS Code's UI is based on JavaScript and HTML, third-party extensions are not allowed to take full advantage of this. Instead, they are only allowed to use what VS Code exposes through public APIs for third-party extensions. In TruffleSqueak, on the other hand, we not only have full control over the UI components in our tools. As a self-sustaining programming system, all parts of the system are accessible and can be changed by the user.

Summary We present two additional case studies that go beyond TruffleSqueak and demonstrate the generalizability and advantages of our approach:

In the first study, we show that our approach can be applied to a polyglot VM based on a composition of RPython interpreters for Python, Ruby, and Squeak/Smalltalk. Since the RPython framework does not provide any means to accommodate multiple languages, we build an appropriate infrastructure and show that the meta-object protocol of Squeak/Smalltalk can be used as a language interoperability protocol. Although the system is not as extensive as TruffleSqueak, it enables exploratory programming across all supported languages and makes it possible to build polyglot-aware tools for polyglot programming. Since we are in full control of the execution model, we also adapt the debugger of Squeak/Smalltalk so that it can be used across all languages, including experimental support for hot-code reloading in Python and Ruby.

The second study illustrates how we build two polyglot notebook systems, transferring the insights we gained through building the `PolyglotNotebook` in TruffleSqueak to create similar experiences in Jupyter and VS Code. When re-using an existing kernel for Jupyter written in JavaScript, the implementation effort is low but access to language interoperability is limited to Graal.js' polyglot API. We conclude that without extending Graal.js, it is hard to build tools for polyglot programming this way because features such as variable sharing can neither be implemented in a language-agnostic nor polyglot-aware way. Furthermore, we show that it is possible to create a polyglot notebook system on top of GraalVM's LSP implementation. While this does allow us to implement language-agnostic and polyglot-aware features, it also requires us to build an extension for VS Code to connect the extended LSP server with its notebook UI. Overall, we find that exploration of ideas for polyglot notebooks is more productive in TruffleSqueak compared with Jupyter or VS Code. As a self-sustaining programming system, it imposes fewer limitations because everything can be easily accessed and changed, which allows us to focus on the aspects we want to explore. Moreover, TruffleSqueak makes it possible to build and evolve a notebook tool at run-time and thus without losing state due to restarting the UI or a kernel. At the same time, our exploration revealed knowledge that can be transferred to the two other systems.

Part VI.

Discussion and Conclusions

14. General Observations and Insights

Throughout the process of conducting our research, we made a number of observations and gained different types of insights into polyglot programming and polyglot VMs. As part of our case studies, we experienced polyglot programming with polyglot VMs from the perspective of both tool and application developers. During the development of TruffleSqueak and Squimera, we acted as runtime and language developers, which helped us to understand various advantages and challenges on the level of runtimes and their language interoperability protocols. In this chapter, we discuss these observations as well as insights and highlight some more general challenges of polyglot programming and polyglot VMs.

14.1. Advantages of Polyglot VMs

First of all, the widespread use of C-based foreign function interfaces, interprocess communication, and language bindings demonstrates that there is a clear need for integrating different languages. This goes so far that developers are even willing to trade run-time performance and tool support for the ability to use more than one language.

Polyglot VMs provide a powerful approach to polyglot programming that comes with many advantages over established approaches: They, for example, provide the necessary infrastructure to facilitate communication between the languages they support. This means that developers no longer have to deal with low-level details of an FFI, serialization of the data structures they would like to share across languages, or other glue code needed for the integration of particular libraries, frameworks, and languages. Instead, they can share complex objects across languages without further ado and focus more on the components they would like to combine. An example of this is our integration of a Ruby library for syntax highlighting into the tools of Squeak/Smalltalk: The integration combines a formatter with different lexers from the library as if they were objects from Smalltalk. Apart from using the polyglot API, no additional glue code was required which would justify the creation of a language binding or wrapper library. In general, we believe that language bindings and wrapper libraries can be replaced with appropriate calls to the polyglot API of a polyglot VM.

Moreover, the direct exchange of objects and messages between languages has another advantage: It can avoid performance overheads caused by serialization as well as data duplication and synchronization problems. Polyglot VMs such as GraalVM have demonstrated that their approach to polyglot programming can be fast [58], for example, by deploying JIT compilers that can perform optimizations across language boundaries.

The high-level integration of languages as provided by polyglot VMs further allows the construction of tools that work across languages. Therefore, developers are not required to switch between tools for different languages. `gdb` and other low-level tools from the OS that usually have no understanding of high-level abstractions of languages but that are often needed to debug FFI calls can be avoided. Furthermore, language interoperability protocols as deployed by some polyglot VMs allow tools to be built in a language-agnostic and polyglot-aware way. This reduces the costs of tool-building as most of a tool's implementation can be shared across languages. At the same time, developers can benefit from a more consistent programming experience, using one familiar set of tools to develop code in multiple languages. With our work, we have shown that polyglot VMs make it possible to re-use the exploratory tools and even the live programming experience of a self-sustaining programming system across languages, which goes beyond the reuse of libraries and frameworks and demonstrates the capabilities and potential of polyglot VMs further.

14.2. Disadvantages of Polyglot VMs

While the approach of polyglot VMs to polyglot programming provides many opportunities for developers, it also has some conceptual disadvantages. Many language implementations are usually maintained on their own and are thus not designed to be re-used in polyglot VMs. Instead, a substantial amount of work is often put into implementing languages from scratch as GraalVM demonstrates: Although it is designed to be a polyglot VM from the start and has seen years of development, only two of its language implementations are officially supported. At the time of writing, those are Graal.js and Sulong. All other GraalVM languages are considered experimental and are not fully compatible with their reference implementations. In a polyglot VM ecosystem with many users, however, it is expected that this work is amortized over time as more code can be re-used with less glue code and as developers benefit from better tool support.

Another problem is that polyglot VMs often require more resources to operate: As our benchmarks in Section 11.2 have shown, TruffleSqueak requires significantly more CPU cycles and memory compared with the

OpenSmalltalkVM. And that is without interacting with any other language. Although additional work can make TruffleSqueak more efficient, GraalVM and Truffle are known to be resource hungry. Nonetheless, we believe there are reasons why other polyglot VMs may face the same problem, albeit maybe on a different scale. One such reason is that language developers are not always free to make certain design decisions because they have been made by the developers of the polyglot VM. The OpenSmalltalkVM, for example, can deploy a JIT compiler designed specifically for Smalltalk and an object layout that is more efficient than that of TruffleSqueak, which must use Java objects to represent Smalltalk objects. Another reason for the increased use of resources is directly caused by polyglot programming: For each language used in a polyglot application, more code is loaded and more memory is required. This is because all languages including their standard libraries must be fully initialized, even if application code only uses a small fraction of them. Although modern computers provide sufficient resources for polyglot VMs, projects such as GraalVM Native Image are actively working on reducing their CPU and memory footprint, for example, through AOT compilation and optimizations based on a closed-world assumption [216].

The fact that polyglot VMs deploy alternative language implementations has further disadvantages: Many programming languages are still evolving. Therefore, alternative implementations in polyglot VMs often require continuous maintenance to stay up to date with the reference implementation or the language specification. At the same time, the protocol for language interoperability of a polyglot VM may also be evolving. Every time this protocol is extended, language implementations as well as tools based on the protocol need to be updated to make use of newly introduced capabilities. Moreover, some programming languages lack proper specifications, which makes it hard to provide full compatibility. Some incompatibilities can also not be avoided due to particular design decisions and assumptions made by the developers of a polyglot VM. All GraalVM languages, for example, must use the GCs from the JVM, which makes it cumbersome to provide compatibility with specific garbage collection algorithms used by corresponding reference implementations. The use of Java has many other implications for language implementations, for example, in terms of the execution model or exception handling. Its poor support for continuations makes it hard to support continuations in guest languages, and doing so also comes at a performance penalty. Moreover, language incompatibilities can hinder the adoption of alternative implementations. GraalPython, for example, does not properly support many Python libraries and frameworks, which is especially limiting when it comes to use cases involving machine learning or data science, two domains that heavily rely on Python.

14.3. Reasoning About Multiple Languages at the Same Time

High-level language interoperability as provided by polyglot VMs makes it easier to combine more software written in different languages. At the same time, combining languages can add additional cognitive overhead in some cases: With every language used in a software system, developers have to think about more syntaxes, semantics, as well as programming concepts, paradigms, and best practices at the same time. Sometimes, however, it is already hard enough to handle of all this when using just one language. During polyglot programming with GraalVM, we observed that even language specifics that are usually simple to think about can lead to scenarios where such specifics need to be carefully considered by developers. Examples of this are differences across languages in terms of object equality, identity, the precision of floating-point numbers and arithmetic operations, type coercion, mutability and immutability, among others.

One way to decrease this cognitive overhead is through maintaining a coarse granularity when writing polyglot applications: We experimented with combining languages on the level of functions, expressions, or even sub-expressions, but found that modules provide the best level for polyglot programming. We believe one reason for this is that software is often split into modules to separate concerns [51, pp. 44–46] and that such concerns provide a good basis for developers to think about different languages. Another indication for this is that the API of a module often plays a more important role than the language it is implemented in, especially when their implementation is treated as a black box. Public APIs allow developers to re-use libraries and frameworks independently from the languages they are written in.

Another way to reduce additional cognitive overhead is through appropriate tool support based on dynamic run-time data as well as through exploratory programming as some of our case studies have shown. Interactive evaluation of code makes it easy to try something out, validate assumptions, and observe and understand the interaction between languages in specific scenarios. Tools that are aware of the polyglot VM can provide additional information about the supported languages, hide polyglot APIs from the developers, and help them to understand how objects and messages are exchanged between languages. For us, the ability to quickly build tools for specific purposes has also proven to be useful in the context of polyglot programming and GraalVM.

14.4. Dealing With Interface and Type Mismatches

Although polyglot VMs allow developers to share complex objects between languages, libraries and frameworks can sometimes not directly be re-used due to interface and type mismatches. For types and interfaces directly supported by the language interoperability protocol of a polyglot VM, portability of code can be improved by mapping types and interfaces between languages automatically. While this works well for primitive values, such as numbers and strings, it is sometimes unclear how complex types should be mapped between languages. In GraalVM, for example, Ruby strings appear as JavaScript strings in JavaScript, exposing the interface of JavaScript strings through its `String` prototype. This is not the case for `Hash` and `Array` objects from Ruby, which the developers of Graal.js decided to expose as normal JavaScript objects with the original interfaces from Ruby. For portability reasons, however, developers may want these objects to use the JavaScript prototypes for `Map` and `Array` instead. So far, GraalVM does not provide any means for developers to control how objects appear in different languages on the guest language level. TruffleSqueak demonstrates a flexible but language-specific solution to this problem: Since it implements the language interoperability protocol from GraalVM on the language level, developers can adjust the behavior of their Smalltalk objects and can control how objects from other languages appear within Smalltalk at run-time. We used this capability, for example, to control how a Smalltalk dictionary is exposed as a key-value store in other languages within the `PolyglotNotebook` presented in Section 12.1. As an alternative, developers can apply the adapter pattern to map different interfaces across languages manually [122]. This, however, only works if identity is irrelevant. GraalVM's target type mappings [142] automate this idea to some extent on the level of its host language. These mappings, however, can only control how objects from the host language are exposed in guest languages.

Another problem is caused by the fact that values of GraalVM guest languages are allowed to have multiple traits and thus, it is not always clear how values should appear across different languages. Language developers could, for example, decide that particular objects of their language expose hash entries, array entries, and possibly other traits all at the same time. It may be possible to expose each of these traits individually in a language. In TruffleSqueak, for example, we experimented with a `ForeignArray` class that developers can request explicitly for any foreign object with the array trait through an `asArray` method, which returns a wrapper for the object that exposes the Smalltalk collection protocol. However, exposing separate types for every possible combination of interoperability traits appears to be impractical, not just because of scalability concerns. For one, most lan-

guages do not support traits, mixins, or multiple inheritance, which could be used to create, for example, a `ForeignArrayDictionaryAndStream` class in TruffleSqueak for representing values with the traits for array elements, hash entries, and buffer elements. This approach can further lead to name clashes between language-specific interfaces for corresponding traits. In Squeak/Smalltalk, for example, `Dictionary` and `Array` have different implementations for the `at:` and `at:put:` methods, which would already cause problems in a `ForeignArrayAndDictionary` class. Consequently, we argue that it makes sense to give developers some control over this, for example, by providing appropriate wrappers through helper methods such as `asArray`, `asDictionary`, and so on. This allows developers to choose a particular trait explicitly in case the default representation does not fit their use case.

The problems, however, are not limited to the level of languages and their implementations. A recurring challenge that we observed while working with our polyglot notebook systems is related to data frames, a data structure commonly used for data analysis. Data frames can be first-class objects of a language, which is the case in R. The data structure can also be defined as part of a third-party framework or library. An example of this is `pandas` [105], a data analysis library written in Python. Its `DataFrame` implementation is inspired by the one in R so one would expect that developers can use the two interchangeably. Although they are conceptually the same, they expose similar but different interfaces and are distinct types. As a consequence, it is not possible to plot a `pandas DataFrame`, for instance, with the `ggplot2` package from R due to failing `is.data.frame` type checks in `ggplot2`. In Section 12.5, we encountered the same problem for a different data type when building a polyglot drawing engine for Squeak/Smalltalk: `Rectangle` objects from Squeak/Smalltalk and Java are not interchangeable because they are of different types and have different interfaces.

Overall, we believe that interface and type mismatches cannot be avoided by language and runtime developers. GraalVM can only provide compatibility between values of different languages on a best effort basis. And some mismatches can only be resolved by application developers and through appropriate wrappers. For this, however, application developers need additional control over the appearance of their objects in different languages, which in turn is an additional responsibility and can lead to additional work.

Summary As the last part of our fifth contribution, we summarize our observations and insights into polyglot programming and polyglot VMs that we gained throughout conducting our research.

Polyglot VMs allow direct communication between languages without the need to serialize, duplicate, or synchronize data and thus reduce the amount of glue code and can make language bindings and wrapper libraries redundant. We have also shown that reuse is not limited to libraries and frameworks. TruffleSqueak demonstrates that polyglot VMs allow us to re-use the exploratory tools and the live programming experience of a self-sustaining programming system for other languages and polyglot programming.

However, polyglot VMs need separate language implementations, which causes a substantial amount of work and can lead to incompatibilities with corresponding reference implementations. They also consume more memory not only because multiple languages run at the same time but also because they must provide infrastructures that are general enough to accommodate different languages. This in turn makes language-specific features and efficient object representations harder to implement.

Moreover, reasoning about multiple languages at the same time can add cognitive overhead for developers. We found that combining code from different languages on the module level can help to deal with this overhead. Similarly, exploratory programming tools can help developers to observe and understand how languages interact with each other in detail and at run-time.

Similar objects from different languages may provide different interfaces and are usually of different types, which both reduce the portability of code across languages. While this can be additional work, we argue that these mismatches can best be mitigated by the user. For this, polyglot VMs must allow developers to control how objects appear in different languages.

15. Related Work

In this chapter, we present five categories of related work: Section 15.1 details related environments for exploratory programming. In Section 15.2, we present related tools designed to support multiple languages. Related work on tools that support developers in building polyglot applications are discussed in Section 15.3, platforms for developing languages and tools in Section 15.4. In Section 15.5, we go into detail about related work on dynamic run-time data and its use in tools.

15.1. Exploratory Programming Environments

Our approach enables exploratory programming in the context of polyglot VMs by re-using the tools of an existing self-sustaining programming system. In the following, we present several other programming systems that support exploratory programming in different contexts.

Self [207] is a programming language, environment, and VM with support for exploratory programming [208]. Its language is dynamic, object-oriented, and uses prototypes instead of classes for inheritance. Its graphical programming environment is self-sustaining and based on the Morphic framework, which was created as part of the Self project. Furthermore, the project led to many advances in JIT compilation techniques that are still used by modern JIT compilers today.

It comes as no surprise that Self has had a strong influence on GraalVM and Squeak/Smalltalk, based on both we built TruffleSqueak. Not only has a lot of the VM work been applied in VMs for Squeak/Smalltalk but also in the JVM on which GraalVM is based. As part of the Klein project, Self's exploratory programming environment was used for research on metacircularity [209], an idea that can also be found in Squeak/Smalltalk [70] and GraalVM [222]. While our approach does not increase the level of metacircularity of polyglot VMs, it enables exploratory programming and tool-building on top of them. Our implementation has shown that exploratory tools can be used not only for user applications. They can also be used for guest languages and internal components of the polyglot VM if they are written in the host language and interoperability with the host language is supported. Extending metacircularity across an entire polyglot VM would, for example, allow the development of

the Graal compiler from within TruffleSqueak and is an interesting direction for future work. Moreover, prior work also explored how Smalltalk and Java can be implemented on top of Self so that they can benefit from the adaptive optimizations of the Self VM [218]. Re-using the tools from the Self environment for Smalltalk or Java and exploring ideas for new ones was, however, not a goal of the work. For Smalltalk, for example, common Smalltalk tools were re-created in Self instead. Our work, on the other hand, primarily focuses on the reuse of existing exploratory tools as well as on exploring tooling ideas for polyglot programming.

Lively Kernel [74] is a self-sustaining programming system for the web with support for live and exploratory programming. It is inspired by Smalltalk, written in JavaScript, builds on an implementation of the Morphic framework, and provides tool-building capabilities. Over the years, Lively Kernel has been used as an exploratory platform for different web technologies but also for native applications as well as for 3D and mobile applications [72]. In Lively4 [96], a successor of Lively Kernel, the abstractions on top of which its exploratory tools work have been lowered from Morphic and a specific JavaScript code structure to HTML DOM and plain JavaScript. This way, Lively4 can also be used as an exploratory programming environment for external web and JavaScript applications that are not directly built within the system.

While Lively Kernel and Lively4 provide exploratory programming and tool-building capabilities for the web, our work provides similar means for polyglot VMs. Similar to Lively4, our approach allows exploratory tools of a self-sustaining programming system to be used for other languages and applications that are outside of the system. This is possible because the tools use the language interoperability protocol of the polyglot VM as an abstraction.

Moreover, and although both re-use concepts from Smalltalk and Morphic, Lively Kernel and Lively4 are built entirely from scratch. Our approach, on the other hand, proposes to re-use an existing SSPS, which TruffleSqueak has demonstrated to be possible.

SqueakJS [47] is a Smalltalk VM written in JavaScript and can run stock Squeak/Smalltalk images in web browsers. It also provides interoperability between Smalltalk and JavaScript through a bi-directional bridge between the guest and the host language. This bridge has been extensively used by Caffeine [92] to use Squeak/Smalltalk as a live and exploratory programming environment on top of web browsers, for example, to explore ideas for virtual reality live-coding spaces in 3D.

To use Squeak/Smalltalk on top of SqueakJS as an exploratory tool-building platform following our approach, a polyglot VM would need to support

JavaScript and web browser capabilities. Since SqueakJS is implemented on top of a high-level language, it faced similar challenges as TruffleSqueak and because of that, inspired several implementation strategies in our implementation. To improve UI performance, for example, SqueakJS also deploys JavaScript ports of BitBlt, Balloon, and other VM plugins.

Poplog [17] is an AI development environment with support for exploratory programming and multiple languages. It uses incremental compilers for Common Lisp, Pop-11, Prolog, and Standard ML and a two-level virtual machine [180] for execution. Furthermore, it provides a screen editor called VED and other programming tools, for example, to incorporate other languages into Poplog.

Poplog mainly focuses on the use and development of languages in the context of AI. While it provides several tools, the lack of a common infrastructure for language interoperability makes it hard to explore and apply tooling ideas that work across all existing and future languages of the system. Our approach requires such a common infrastructure to provide exploratory programming and tool-building capabilities on top of polyglot VMs.

15.2. Dynamic Tools With Multi-Language Support

Tools play a central role in our work. For polyglot programming, tools must support multiple languages at the same time. This section presents related work on tools and tooling infrastructures that are designed for this.

Multi-Language Debugging is among the most researched topics in the field of polyglot programming. Numerous approaches have been explored in several polyglot systems (e.g., [95, 106, 125, 193, 210, 211]). Debuggers are important programming tools and because of that, their general features and requirements are mostly well-understood. This, however, is not true for their implementations, which heavily rely on the execution environments that they target.

While debugging is also crucial for polyglot programming, our work aims at exploring new tools as well as features of tools that support developers in writing polyglot applications in entirely new ways. Nonetheless, the requirements of debuggers and exploratory tools overlap. Both, for example, provide means for object inspection. We, therefore, believe that implementations of our approach can benefit from efforts to support multi-language debugging in a polyglot VM.

Truffle's Instrument API [210] provides a language-agnostic infrastructure for instrumenting GraalVM languages. This API operates on the AST level and allows tools to perform operations before, instead of, or after the execution

of specific AST nodes. For this, appropriate wrapper nodes are inserted into an AST for all instrumented nodes. The Instrument API makes it possible to implement debuggers, profilers, and similar tools in such a way that they work across all GraalVM languages. It can further be used to collect dynamic run-time data for dynamic program analysis [195] and to build tools that support developers in building software with one or more languages (see Section 15.5).

While our research builds on the idea of language-agnostic tool-building, it is different to the work on the Instrument API in four ways:

1. The language-agnostic approach to tool-building allows tools to be built once and re-used across all languages of a polyglot VM. To better support polyglot programming, we extend the idea of language-agnostic tools and propose to make them aware of the polyglot environment. Only if tools are *polyglot-aware*, they can, for example, hide polyglot APIs from the users and help them to distinguish between objects from different languages.
2. Instead of treating tools as external actors that must communicate through a possibly restrictive interface for tools, our approach allows tools to be built on the guest language level at run-time and to communicate directly with applications through the language interoperability protocol. From the perspective of the GraalVM, there is no difference between tools built in TruffleSqueak and user applications.
3. We showed that existing tools can be adapted so that they work across different languages of a polyglot VM. GraalVM's debugging and monitoring infrastructure, on the other hand, was entirely built from scratch with the Instrument API.
4. And lastly, we demonstrated that polyglot programming can be applied to tool-building, not only to make tool-building more productive but also to gain further insights into polyglot programming itself.

15.3. Tools for Building Polyglot Applications

As part of Chapter 12, we have used an implementation of our approach to explore tooling ideas that help developers to build polyglot applications. In the following, we present some related tools and systems.

Some IDEs support developers in building applications using multiple languages. IntelliJ IDEA [77], for example, not only supports Java but also other JVM-based languages, such as Kotlin, Scala, or Groovy, that can be used within the same project. Visual Studio [112] allows developers to combine multiple .NET languages in a similar way.

IDEs like this provide well-understood tools such as code editors and debuggers and often support a specific set of languages. They, however, commonly run separately from the runtime, and therefore, their tools are restricted to the capabilities of the runtime's tool interface. At the same time, such IDEs are often complex, proprietary, and provide restrictive plugin systems for extensions. Our approach, on the other hand, allows tools to run and to be built in the same runtime process that also executes user applications. Since we propose to re-use a self-sustaining programming system, tools can freely be built and evolved just like the rest of the system.

Polyglot Notebook Systems allow data scientists and researchers to use multiple languages within the same notebook [94]. SoS Notebook [148], BeakerX [206], and many other systems orchestrate multiple notebook kernels for different languages through inter-process communication. Polynote [116] supports Scala and Python by embedding the CPython interpreter through the Java Native Interface. .NET Interactive Notebooks [109], on the other hand, provides access to and interoperability between .NET languages within a notebook through a single, .NET-based kernel.

In terms of language integration, the polyglot notebook systems we presented in Section 12.1 and Section 13.2 can best be compared with .NET Interactive Notebooks: All of them are based on polyglot VMs. Nonetheless, the .NET-based kernel is with approximately 2900 SLOC also more complex compared with the 725 SLOC required to build the entire notebook system in TruffleSqueak. This is therefore another example of how our approach allows the exploration of specific tooling ideas with relatively low costs.

Eco [35] is an editor for building polyglot applications and allows fine-grained language compositions. For this, it uses an incremental parser with support for different languages to maintain an AST-based representation across all languages it supports. Moreover, the parser has been extended to automatically detect different languages [34].

Eco is a static tool and as such, does not have access to dynamic run-time data. To support another language, its parser must be extended appropriately. Also, the execution of polyglot applications is not a primary concern of Eco. Our polyglot editor presented in Section 12.2, on the other hand, specifically integrates the polyglot APIs of GraalVM languages to allow the execution on top of the GraalVM. Nonetheless, the work on Eco has served as an inspiration for the code boxes supported in the polyglot editor. While Eco supports both implicit and explicit switches between languages, code boxes in our editor can only be added explicitly by the user.

15.4. Platforms for Language and Tool Development

Language and tool development often go hand in hand, not just when it comes to polyglot VMs. There are many different platforms that enable both and we present some that are related to our work in the following.

Helvetia [158] is a Smalltalk-based environment for language development. Smalltalk is used as both the development environment and the host language for building embedded languages. DSLs and other languages implemented in Helvetia are represented using Smalltalk ASTs. This allows different tools from Smalltalk, such as its code browser, parser, or debugger, to be re-used across languages.

Helvetia and our work share the same idea: re-using existing tools across different languages. Helvetia does that for languages embedded in Smalltalk, where languages are integrated vertically, and focuses on language development. Our approach, on the other hand, enables exploratory programming and focuses on tool-building for polyglot VMs, where guest languages are integrated horizontally, so on the same level.

Gramada [154] is another Smalltalk-based environment for language development. It builds on the Ohm parser generator [212] and aims at enabling live programming in language development. For this, it provides immediate and continuous feedback through, for example, syntax tests that run automatically, an interactive debugger, and a visualization tool for parse trees.

While our approach does not aim at the live development of languages, it enables exploratory tools to be used by language developers of polyglot VMs to explore their languages at run-time. These tools support live programming to some extent. As our implementation has shown, GraalVM languages can be explored with TruffleSqueak but their implementation cannot be changed arbitrarily at run-time as they are written in Truffle and Java.

Language Workbenches [46], such as Spoofax [82], Xtext [39], or JetBrains MPS [78], are related platforms for language development. While they are designed to support language developers to create and evolve languages, many workbenches also provide means to generate different static tools, such as syntax highlighting, code navigation, or refactoring, from language definitions [38].

The primary goal of language workbenches is to support the development of new languages. Tool development, which is often limited to static tools, and language interoperability are often secondary concerns. Instead of focusing on language development, our approach makes exploratory programming tools available across different languages and allows the construction of both static and dynamic tools at run-time.

15.5. Dynamic Run-Time Data and Tools

Dynamic tools have demonstrated that dynamic run-time data can be used to support developers in writing code in dynamic programming languages. In this thesis, we argue that such tools can also support them during polyglot programming with polyglot VMs because language interoperability is dynamic and can best be observed at run-time. In general, two main aspects are often researched in related work: The collection of dynamic run-time data and how this data can be put to use within dynamic tools.

Hermion [167], for example, is an IDE based on Squeak/Smalltalk and uses partial behavioral reflection [198] to obtain dynamic run-time data on the language level. Based on this information, tools for code navigation can be improved and source code can be annotated with type information to aid program comprehension.

Senseo [166] and work by Holmes and Notkin [67] use Aspect-Oriented Programming (AOP) [85] through AspectJ [84] to collect dynamic trace data, which they then incorporate into Eclipse. Similar to Hermion, the data is used to annotate code with type information and to improve the precision in the "find references" tool from Eclipse.

The work on *Type Harvesting* [62], on the other hand, is primarily concerned with the collection of type information at run-time. It is based on the stepwise execution of code to observe and collect values and their concrete types, again, during test execution.

Live Typing [215], on the other hand, collects dynamic type information on the VM level. Its implementation is based on Cuis Smalltalk and a modified OpenSmalltalkVM that stores type information during the execution of specific bytecodes in data structures that can be accessed from within Cuis. The collected dynamic run-time data is again used to improve code navigation but also to improve code completion and to provide better refactoring tools.

Truffle's Instrument API [210] allows dynamic program instrumentation, which can also be used to collect dynamic run-time data for tools supporting developers in building applications with one or more languages. In related work, we demonstrated that is possible to build an LSP server based on this that provides, for example, code completion or go-to definition across GraalVM languages [192]. In addition, we showed that an example-based live programming system can be built on top of such an LSP server [128].

In self-sustaining programming systems, reflective facilities always provide access to dynamic run-time data. Our approach, however, shows up another way to acquire dynamic run-time data: Instead of collecting such data yourself, data already collected on the VM level can be accessed and re-used. Language implementations and JIT compilers, for example, manage

and collect a lot of data at run-time. This often includes profiling information, numerous caches such as polymorphic inline caches [68], state for different performance optimizations, and IRs of the compiler infrastructure among others. In TruffleSqueak, we have shown that it is possible to access profiling and other information collected within GraalVM languages or the Graal compiler. Graal's call target objects, for example, provide access to invocation counts as well as type information for arguments and return values of methods. In Section 12.4, we have further presented two tools that help not only runtime and language developers to understand the behavior of the Graal compiler at run-time. The `CallTargetBrowser` can also be used by application developers to find, for example, unused code or performance issues within their applications. The exploratory tools from TruffleSqueak and its VM introspection capabilities allow further exploration of other GraalVM internals to identify other useful sources of dynamic run-time data.

16. Conclusions and Future Work

This chapter presents directions for future work and concludes the thesis.

16.1. Future Work

Future work can be grouped into four areas: technical limitations, opportunities to build on our work on tools, challenges of polyglot VMs, and best practices for polyglot programming.

Limitations of TruffleSqueak and GraalVM As discussed in Section 11.4, TruffleSqueak has several limitations and raised some challenges that are specific to GraalVM and Squeak/Smalltalk. These could be addressed in future work. GraalVM's language interoperability protocol could be extended with our optional features for exploratory programming such as `findAllInstances` or `swap` to allow them to work across all GraalVM languages. It would also be interesting to enable cross-language debugging from within TruffleSqueak, for example, by running it or only its UI process in a separate Java thread. Moreover, its run-time performance and memory footprint could be improved further with more efficient object representations, additional optimizations, or through AOT compilation with GraalVM Native Image. TruffleSqueak could also be extended with more plugins, primitives, and other missing VM features to increase its compatibility and to allow other dialects or descendants of Smalltalk such as Cuis or Newspeak to run on GraalVM.

Taking Our Work on Tools Further Future work could also build on the results of our case studies from Chapter 12 and Chapter 13 and push these ideas forward. It could also use TruffleSqueak to explore additional ideas for tools that help developers, for example, to deal with the cognitive overhead of polyglot programming or to find and combine appropriate languages, frameworks, and libraries when building polyglot applications. Before our and other ideas can be applied in practice and turned into products, it makes sense to evaluate them further with appropriate user studies. In addition, our approach could also be applied to .NET and other polyglot VMs. Similarly, other self-sustaining programming systems such as Self, Cuis, or Lively Kernel could be investigated instead of Squeak/Smalltalk.

Challenges of Polyglot VMs Moreover, we discussed several advantages, disadvantages, and challenges of polyglot VMs in Chapter 14 that are not specific to GraalVM and could also be the subject of future research. Future work could investigate how type and interface mismatches across languages can be mitigated to further improve the portability of code. Another interesting challenge to solve is how different languages can co-exist and interoperate with each other without breaking each other's models for exception handling, continuations, and other mechanisms beyond call-and-return control flow. The same is true for combining different programming paradigms and other language concepts, such as inheritance, garbage collection, or object persistence, across languages. An additional direction for future work is to research how the idea of self-sustainability could be expanded across languages and a polyglot VM so that all components of a polyglot VM's ecosystem can be evolved through itself at run-time. In the case of TruffleSqueak, this would allow the development of performance optimizations for the Graal compiler with shorter feedback loops.

Toward Best Practices for Polyglot Programming Best practices for polyglot programming with polyglot VMs are needed to make it more approachable in practice. Therefore, future work also needs to research other aspects of software engineering in the context of polyglot programming. It could examine the impact of this type of programming on the readability and maintainability of code as well as on developer productivity. It could also focus on software design patterns, software testing, software maintenance, dependency management, and other software development activities, practices, and methodologies with regard to building polyglot applications.

16.2. Conclusions

Polyglot VMs provide a new level of polyglot programming that comes with many advantages over C-based foreign function interfaces, inter-process communication, and other established approaches. Through high-level language interoperability protocols, they allow languages to directly interact with each other. This allows developers to re-use and combine libraries and frameworks written in different languages with less glue code and lower performance overheads compared with other approaches. At the same time, it is possible to build tools that support developers across multiple languages. While prior research has investigated how debuggers and other conventional tools can be built for polyglot VMs, our work focuses on novel tools that are specifically designed to support developers in building polyglot applications.

> **Contribution 1** An approach enabling exploratory programming and tool-building on top of polyglot virtual machines.

In this thesis, we presented an approach for an exploratory tool-building platform for polyglot VMs. With exploratory programming tools, developers can build, evolve, inspect, and interact with their applications at run-time, allowing them to explore and gather software requirements. Such tools, however, are usually built for and limited to a particular programming language ecosystem.

Our approach builds on a self-sustaining programming system and shows that it is possible to re-use existing tools for exploratory programming. For this, we propose to integrate other languages of a polyglot VM into such a programming system based on the VM's protocol for language interoperability. This way, the exploratory tools require little to no modification and work across the existing and all future languages supported by a polyglot VM. We described the API requirements for such an integration, which overlap in large part with what polyglot VMs already have to support for language interoperability. Although an SSPS needs to run as a guest language of a polyglot VM for this, we argue that the required language implementation is usually small compared to the rest of the system.

In return, it is possible to re-use and build on the entire SSPS including its standard library, UI system, features such as live programming, and especially its extensive tool-building capabilities. Development in such programming systems happens at run-time. Therefore, dynamic run-time data is always accessible for the construction of dynamic tools. Since the language interoperability protocol is directly exposed within the SSPS for the exploratory tools, we showed that it can also be used to extend other tools and to build new ones that work across multiple languages.

> **Contribution 2** Extensions for exploratory programming tools that make them polyglot-aware.

While language-agnostic tools as proposed by prior research are a good step toward providing better tool support across languages, we argue that developers need more than language-agnostic views to build polyglot applications. Tools must be *polyglot-aware* so that they support developers with features specifically designed for polyglot programming and polyglot VMs. This also applies to the exploratory tools that our first contribution enables. Therefore, we propose extensions that make these tools aware of, and with that even more useful for polyglot VMs. In addition to the structure of an object, inspection tools can further help developers by revealing the interface of an object and its language of origin. Exploratory tools that are aware of the polyglot

VM they run on can also provide a better user experience, for example, by hiding the polyglot API from users. By incorporating additional features of a polyglot VM's language interoperability protocol into these tools, we showed that they can also support language and runtime developers in understanding dynamic behavior within, and requirements of their polyglot VM.

> **Contribution 3** A proposal to further explore polyglot programming by applying it to tool-building and within a self-sustaining programming system.

We further propose to apply polyglot programming to tool-building to increase the productivity of tool developers. More importantly, this helps to gain new insights into polyglot programming with polyglot VMs and to identify potential use cases and challenges, which our case studies have demonstrated. We showed that polyglot programming can also be applied for the same purposes by application, language, and runtime developers and even within the components of a self-sustaining programming system. That way, the ecosystem of a polyglot VM not only benefits from the capabilities of an SSPS, the SSPS can also benefit from the polyglot VM and its supported languages.

> **Contribution 4** An implementation of said approach and extensions for the GraalVM and based on Squeak/Smalltalk.

Furthermore, we presented TruffleSqueak, an implementation of our approach for the GraalVM and based on Squeak/Smalltalk. TruffleSqueak not only shows that our approach is feasible but also that the implementation in itself helps to test and validate various assumptions, design decisions, and APIs of GraalVM. Although its language implementation is small compared to the rest of TruffleSqueak, some language features of Squeak/Smalltalk, such as its `become:` or snapshotting mechanisms, require appropriate implementation strategies in Truffle. Even without the integration of other languages, we were and still are able to generate useful feedback for the evolution of GraalVM based on implementation challenges and our experiences using TruffleSqueak. Our UI performance benchmarks, for example, illustrate that the latency mode of the Graal compiler is to be preferred over the default throughput mode for TruffleSqueak's programming system, because short warmup times are more important than peak performance for UI applications. Our benchmarks thus help to assess the impact of new compiler features that improve warmup or performance in GraalVM.

With our integration of GraalVM's language interoperability protocol into TruffleSqueak, we can observe how GraalVM languages interoperate with

each other in detail and at run-time. This helped us not only to uncover many inconsistencies and bugs across different GraalVM languages but also to provide further feedback for the GraalVM team and discuss how the protocol could be evolved. An example is an API hook for determining the language of an object, a simple but important feature required for building polyglot-aware tools. We further demonstrated that TruffleSqueak and its tools can be extended through polyglot programming. Examples are a Ruby library that we integrated to provide syntax highlighting for hundreds of languages within the exploratory tools as well as an R package that we combined with Squeak/Smalltalk's Morphic framework to provide live data visualizations.

Overall, TruffleSqueak shows that polyglot VMs allow more than libraries and frameworks to be shared across different languages. It also enables the reuse of tools including those for exploratory programming across languages and brings the programming experience from Squeak/Smalltalk including live programming to polyglot VMs. We believe this is a good demonstration of how much more capable polyglot VMs are compared with established approaches to polyglot programming.

> **Contribution 5** Case studies that demonstrate how our approach enables further research on tools for polyglot programming and polyglot virtual machines, as well as a synthesis of our findings.

Moreover, we presented five case studies that show how TruffleSqueak can be used to explore different research questions on tools for polyglot programming. By composing workspaces and inspection tools, for example, we quickly created a Jupyter-like notebook system that allows data scientists and researchers to use more than one language at the same time. All tools we presented are themselves built in a polyglot way, which allowed us to gain further insights into polyglot programming and how it can be applied when building tools. In an additional study, we demonstrated that the insights gained through the notebook system in TruffleSqueak are transferable to other notebook systems such as Jupyter and VS Code notebooks. In another study, we further showed that our approach can also be applied to an RPython-based polyglot VM.

We summarized the insights and lessons learned for each of our case studies and present a synthesis of our findings. Overall, we believe these studies give a good impression of the potential of polyglot VMs. These VMs allow developers to combine high-level languages without having to deal with low-level abstractions, which increases reuse and thus developer productivity. Our extended exploratory tools and other polyglot-aware tools can help developers to deal with the additional cognitive overhead introduced by polyglot programming. We found that the extent of this overhead not only

depends on the number of languages in use but also on the granularity of their combination. The finer languages are combined, such as on the level of functions, the harder it is, for example, to keep track of the semantics of different languages. Modules, on the other hand, allow developers to think about APIs rather than about implementation details such as the language they are written in and thus provide a reasonable granularity for polyglot programming. Other challenges resulting from high-level language integrations are type and interface mismatches across languages. These mismatches can cause additional work for developers and need to be addressed to make more code portable and with that easier to re-use across languages.

> **Thesis Statement** To use and evolve polyglot virtual machines effectively, we must be able to explore tooling ideas, polyglot applications, language implementations, and the VMs themselves at run-time.

Language interoperability as provided by polyglot VMs can best be observed at run-time. Many developers, however, distinguish between development-time and run-time. Self-sustaining programming systems such as Squeak/Smalltalk, Lively Kernel, or Self have demonstrated that development at run-time can improve productivity through shorter feedback loops. At the same time, dynamic run-time data is always accessible, which can be used in tools to better support developers when writing code in dynamic programming languages. Our approach brings this idea to polyglot VMs and helps tool, application, language, and runtime developers to use and evolve them effectively.

[|]

Developers strive to improve their productivity. Yet they usually limit themselves to a single programming language when building applications. Polyglot programming allows them to use multiple languages but is often only used out of necessity. Established approaches to polyglot programming come with significant limitations, for example, in terms of tool support, which can make for a poor programming experience. Polyglot VMs can avoid many of these limitations but they are also relatively new and not yet well-understood.

In this thesis, we presented an approach for a platform that enables exploratory programming and tool-building in the context of polyglot VMs. On top of such a platform, we further showcased several polyglot applications and tools for building them, demonstrating both the potentials and challenges of polyglot VMs. With good tool support, we believe that polyglot programming with polyglot VMs can be made more approachable in practice. Ultimately, we think that polyglot VMs have the potential to turn polyglot

programming from a necessity into a real opportunity for developers, making it much more common for them to take advantage of multiple languages and their ecosystems when building software.

Part VII.

Appendix

Appendix A.

Bytecode Interpreter Loop Implementations

In this chapter, we present an implementation of a bytecode interpreter loop for Squeak/Smalltalk in Truffle, GraalVM's language implementation framework that is originally designed to implement AST interpreters. We further explain how this implementation needs to be extended so that the Graal compiler can detect and optimize it well.

Listing A.1 shows a simple Truffle node implementation to interpret sequences of Squeak/Smalltalk bytecodes. This node extends Truffle's `Node` class and contains appropriate nodes for each bytecode of the target Squeak/Smalltalk method as child nodes (lines 2 and 6). Within the `executeLoop()` method, the `pc` (program counter) is set to zero to select the first bytecode of the corresponding Smalltalk method before the interpretation loop is entered. Within the loop, the bytecode node for the current `pc` is fetched (line 12). If it is a `ReturnNode`, the loop is left returning the result of the node's

Listing A.1: A simple Truffle node implementation for interpreting sequences of Squeak/Smalltalk bytecodes represented as AST nodes.

```
1  class SimpleExecuteBytecodeNode extends Node {
2    @Children AbstractBytecodeNode[] bytecodeNodes;
3
4    SimpleExecuteBytecodeNode(CompiledCodeObject smalltalkMethod) {
5      super();
6      bytecodeNodes = smalltalkMethod.toBytecodeNodes();
7    }
8
9    Object executeLoop(VirtualFrame frame) {
10     int pc = 0;
11     while (true) {
12       AbstractBytecodeNode node = bytecodeNodes[pc];
13       if (node instanceof ReturnNode) {
14         return ((ReturnNode) node).executeReturn(frame);
15       } else {
16         pc = node.executeInt(frame);
17       }
18     }
19   }
20
21   // ...
22 }
```

`executeReturn()` method (line 14). Otherwise, the `executeInt()` of all other `AbstractBytecodeNodes` is executed (line 16), which performs the appropriate bytecode operation and returns the next `pc` for the next iteration of the loop. Although this simple node implementation can correctly interpret a sequence of bytecode nodes, the Graal compiler requires additional hints to produce efficient machine code for it.

Listing A.2 shows the loop after extending it with hints for the Graal compiler. While this is still a slightly simplified version of the loop used in TruffleSqueak, it covers the important parts that allow the compiler to produce efficient machine code for our bytecode interpreter. The most important hint is the `@ExplodeLoop` annotation of the kind `MERGE_EXPLODE` in line 9. This loop explosion kind is specifically designed for bytecode interpreter loops as it instructs the compiler to explode the loop in the annotated method while merging all copies of the loop body with identical state. To avoid any confusion in the compiler, we mark the `VirtualFrame` parameter, the `node` variable, and others as final, label the while loop, and continue to that label specifically from all loop ends (lines 22, 25, 33, 37, and 40). Instead of returning in case of a `ReturnNode`, the `pc` is set to a well-known negative value and checked in the while condition in line 15. We also assert that the number of `bytecodeNodes` and the `pc` per loop iteration are constant during partial evaluation (lines 11 and 16). This helps to ensure that the `MERGE_EXPLODE` loop explosion kind is applied correctly. Furthermore, we have to help the compiler to detect loops within bytecode sequences by counting back jumps. For this, we introduce a `backJumpCounter` in line 14, which is incremented in case the `successor` is less or equal to the current `pc` (line 30). This can only happen in the case of an unconditional jump bytecode (lines 27 to 33), which the Squeak/Smalltalk compiler uses to generate loops. We also only increment the counter in the interpreter to avoid the profiling overhead in compiled code. Before the `executeLoop()` method returns, we assert that the `backJumpCounter` did not overflow (line 43) and report it to the compiler through `LoopNode.reportLoopCount()` (line 44). Additionally, we further help the compiler to understand how often a conditional jump bytecode jumps or not by profiling the result of `executeCondition()` with a `CountingProfile` from Truffle (line 20). Internally, this type of profile reports the probability of a branch to the compiler. This information is then used, for example, to decide whether the bytecode nodes behind a certain branch should become part of compiled code or not and in which order.

Moreover, Listing A.2 also shows an optimization: Since each bytecode node usually has additional child nodes, ASTs of bytecode interpreters implemented this way in Truffle tend to be rather large compared with normal AST interpreters. The larger an AST, the more memory is consumed

Listing A.2: The node from Listing A.1 extended with hints and an optimization for the Graal compiler.

```
class ExtendedExecuteBytecodeNode extends Node {
  @Children AbstractBytecodeNode[] bytecodeNodes;

  ExtendedExecuteBytecodeNode(CompiledCodeObject smalltalkMethod) {
    super();
    bytecodeNodes = smalltalkMethod.createEmptyBytecodeNodesArray();
  }

  @ExplodeLoop(kind = ExplodeLoop.LoopExplosionKind.MERGE_EXPLODE)
  Object executeLoop(final VirtualFrame frame) {
    CompilerAsserts.partialEvaluationConstant(bytecodeNodes.length);
    int pc = 0;
    Object returnValue = null;
    int backJumpCounter = 0;
    bytecode_loop: while (pc != LOCAL_RETURN_PC) {
      CompilerAsserts.partialEvaluationConstant(pc);
      final AbstractBytecodeNode node = fetchNextBytecodeNode(pc);
      if (node instanceof ConditionalJumpNode) {
        final ConditionalJumpNode jumpNode = (ConditionalJumpNode) node;
        if (jumpNode.profile(jumpNode.executeCondition(frame))) {
          pc = jumpNode.getJumpSuccessor();
          continue bytecode_loop;
        } else {
          pc = jumpNode.getNoJumpSuccessor();
          continue bytecode_loop;
        }
      } else if (node instanceof UnconditionalJumpNode) {
        final int successor = ((UnconditionalJumpNode) node).getJumpSuccessor();
        if (CompilerDirectives.inInterpreter() && successor <= pc) {
          backJumpCounter++;
        }
        pc = successor;
        continue bytecode_loop;
      } else if (node instanceof ReturnNode) {
        returnValue = ((ReturnNode) node).executeReturn(frame);
        pc = LOCAL_RETURN_PC;
        continue bytecode_loop;
      } else {
        pc = node.executeInt(frame);
        continue bytecode_loop;
      }
    }
    assert backJumpCounter >= 0;
    LoopNode.reportLoopCount(this, backJumpCounter);
    return returnValue;
  }

  AbstractBytecodeNode fetchNextBytecodeNode(final int pc) {
    if (bytecodeNodes[pc] == null) {
      CompilerDirectives.transferToInterpreterAndInvalidate();
      bytecodeNodes[pc] = insert(smalltalkMethod.createBytecodeNodeAt(pc));
      notifyInserted(bytecodeNodes[pc]);
    }
    return bytecodeNodes[pc];
  }

  // ...
}
```

but also more time is needed to process it in the compiler, which in turn results in longer warmup times. To reduce AST sizes and with that memory footprint and warmup times, we allocate an empty `AbstractBytecodeNode[]` in line 6. Instead of directly reading from this array, bytecode nodes are fetched in line 17 via `fetchNextBytecodeNode()`. This helper method inserts bytecode nodes on demand. For this, it checks whether the bytecode node for a given pc already exists in the array and simply returns it if it does. Otherwise, it creates an appropriate node and inserts it correctly into the AST (line 51). Since this can happen as part of compiled code, we must instruct the Graal compiler to transfer to the interpreter and invalidate any compiled code for this node (line 50) before modifying the AST. After inserting the new node, we must also notify the Truffle framework about the operation (line 52) to ensure that instrumentation works correctly. Before the Graal compiler optimizes a Smalltalk method, the method is executed in the interpreter and thus it is likely that all required bytecode nodes are allocated when JIT compilation occurs. Since the compiler treats the `AbstractBytecodeNode[]` as final at compilation time due to the `@Children` annotation from Truffle, the `fetchNextBytecodeNode()` helper can be completely optimized away in compiled code. As a result, only nodes for reached bytecodes are allocated, which can reduce AST sizes significantly, for example, when methods contain bytecodes for platform-specific code that will never be executed. On the other hand, this means that as new bytecodes become reachable, compiled code must be thrown away and re-optimized, which is additional work for the compiler.

Although the extended version of the loop is around four times larger in terms of SLOC than the simple loop, it is still reasonable in terms of implementation complexity. However, the process of applying the right hints in the right places is not straightforward and requires good knowledge about both the Graal compiler and the language in question. In Espresso and GraalWasm, the GraalVM team has explored a different implementation strategy for bytecode interpreters that implements the loop and all bytecode operations in a single node to further reduce interpreter overhead and memory footprint. In the future, it would be great if the Truffle framework helps language developers by providing appropriate infrastructures and components for building fast and efficient bytecode interpreters for the GraalVM.

Appendix B.

Language Performance Evaluation

In Section 11.2, we evaluate the UI performance of TruffleSqueak to show that its programming system is usable on the GraalVM. Since these UI performance benchmarks rely on the Morphic framework, they not only exercise a lot of Smalltalk code but also the BitBlt primitives of the corresponding VM as well as platform-specific code for rendering the display buffer from Squeak/Smalltalk. To measure the run-time performance of the language and to compare the effect of different GraalVM modes in more detail, we run the Are We Fast Yet benchmark suite [102], a set of macro and micro benchmarks, on different TruffleSqueak configurations as well as on the OpenSmalltalkVM. We also use this benchmark suite for continuous performance tracking of TruffleSqueak: During development, the benchmark suite is frequently run against new changes of TruffleSqueak, and the benchmark results are added as comments to the corresponding commits on GitHub. This helps us to understand how changes affect the run-time performance and thus allows us to detect and avoid performance regressions.

Setup All benchmarks run on the same dedicated benchmark server that we set up for continuous performance tracking and that we also use in the second UI benchmark from Section 11.2: a Dell PowerEdge 2950 (Two Quad-Core Intel Xeon E5410 CPUs @ 2.33 GHz, 32.18 GiB ECC memory) running on Debian 9. Hyper-threading, Intel Turbo Boost, and Intel P-States are disabled. We use ReBench, a benchmark runner maintained as part of the AWFY project, to execute each benchmark 250 times on each VM configuration: TruffleSqueak on the JDK11-based distribution of GraalVM 21.2.0 CE — the community edition of GraalVM — in latency mode, in throughput mode, on the JDK11-based GraalVM 21.2.0 EE — the commercial enterprise edition — in both modes, and on the OpenSmalltalkVM 202003021730 (64-bit). All configurations use the default problem size for each benchmark and the same prepared image based on Squeak6.0alpha-20089-64bit with the Sista bytecode set and `FullBlockClosures` enabled.

The command that ReBench uses to run the AWFY benchmarks on TruffleSqueak is shown in Listing B.1. The image startup routine and interrupt

Listing B.1: The command used to run AWFY benchmarks on TruffleSqueak.

```
bin/trufflesqueak --experimental-options \
  --smalltalk.disable-startup        \ # skip the image startup routine
  --smalltalk.disable-interrupts     \ # disable the interrupt handler
  --engine.Mode="${GRAALVM_MODE}"    \ # set the corresponding GraalVM mode
  --engine.MultiTier=false           \ # disable multi-tier compilation
  --engine.DynamicCompilationThresholds=false \ # disable dynamic thresholds
  --engine.TraceCompilation          \ # trace compilation of Smalltalk methods
  --engine.CompilationStatistics     \ # print compilation statistics on exit
  --log.file="${BENCHMARK_NAME}.log" \ # redirect logging into a file
  --code "FileStream startUp: true. Harness new run: #(nil '${BENCHMARK_NAME}'
  ↪  250 ${PROBLEM_SIZE})"           \ # Smalltalk code to execute the benchmark
  AWFY.image                           # prepared AWFY Smalltalk image
```

handler are disabled to reduce the amount of additional code executed before and when a benchmark runs. Multi-tier compilation is disabled to ensure that benchmark code is only optimized in the highest tier. We also disable dynamic compilation thresholds to avoid additional compilations, for example, of the benchmark harness after the compilation of the actual benchmark is done. For analysis purposes, compilation traces and statistics are enabled and logged into a file. We use these traces to check that each benchmark stabilizes within the first 50 iterations that we use for warmup. Although this is a fixed number of iterations, the Graal compiler has more time to optimize benchmarks that need longer to warm up and run. Since we disabled the startup routine, the `FileStream class>>startUp:` method needs to be manually invoked to ensure that the `stdio` handles are set in TruffleSqueak before the benchmark harness can report timing information via `stdout`.

Note that, due to stack overflow errors, we excluded the *CD* benchmark, a simulation of a collision detector, from our performance tracking infrastructure and disabled it for this performance evaluation. After this thesis was submitted, we found out that these errors were caused by a bug in the Smalltalk port of the benchmark. The bug resulted in excessive stack depths, which exceeded the default thread stack size of GraalVM and thus caused the *CD* benchmark to fail on TruffleSqueak.

Results Figure B.1 shows plots for 200 iterations of the AWFY benchmarks for all five VM configurations after 50 iterations of warmup. The configurations are sorted in descending order by the total time to run all benchmarks. Table B.1 lists corresponding mean values, 95 % confidence intervals, factors relative to the OpenSmalltalkVM, and geometric means of these factors. To derive the mean values and confidence intervals, we used the bootstrapping technique described in [81].

The four macro benchmarks — *DeltaBlue*, *Havlak*, *Json*, and *Richards* — show that TruffleSqueak on GraalVM in latency mode is slower than in throughput mode, which is expected and in line with our observations in Section 11.2.

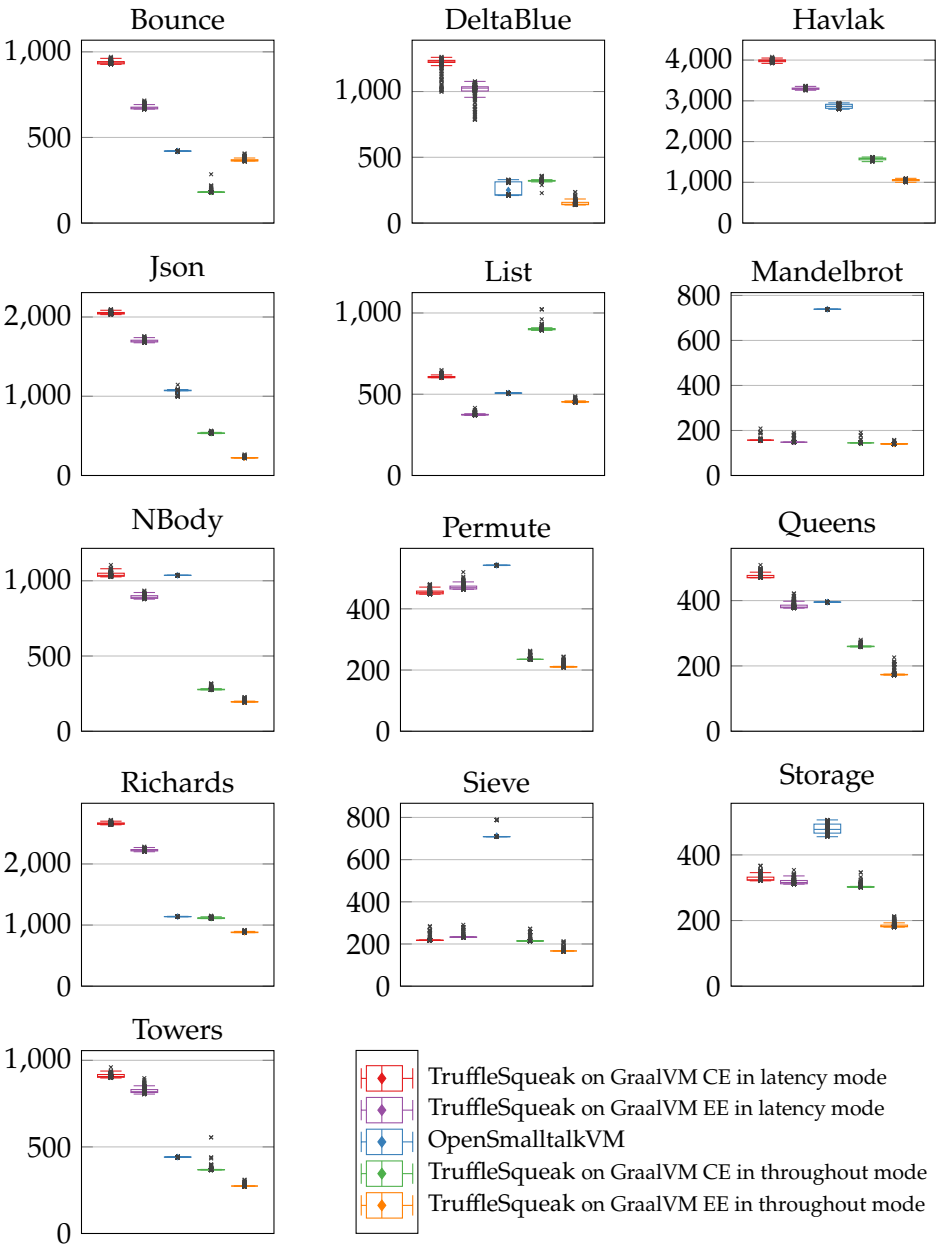


Figure B.1.: Peak performance plots for the AWFY benchmarks in milliseconds per VM configuration sorted by the total time to run all benchmarks. Lower is better. Each Tukey boxplot [104] visualizes the quartiles of 200 iterations per VM (after 50 iterations of warmup). The median is highlighted with a diamond marker. The ends of the whiskers represent the lowest and highest datum within 1.5 IQR of the corresponding quartile.

Table B.1.: Peak performance result table for the AWFY benchmarks shown in Figure B.1. For each benchmark, the first row shows absolute mean values and 95 % confidence intervals for 200 iterations and after 50 iterations of warmup. The second row shows those values relative to the OpenSmalltalkVM as a factor. The best and worst benchmark results are underlined in green and red respectively. The last row contains the geometric mean of the relative factors

Benchmark (*problem size*)	**TruffleSqueak** *in* **Latency Mode**		**OpenSmalltalkVM**	**TruffleSqueak** *in* **Throughput Mode**	
	GraalVM CE	*GraalVM EE*		*GraalVM CE*	*GraalVM EE*
Bounce (1500)	938.270 ms±1.063	674.580 ms±1.217	420.218 ms±0.017	182.695 ms±1.116	368.605 ms±1.082
	2.233±0.003	1.605±0.003		0.435±0.004	0.877±0.003
DeltaBlue (12000)	1209.120 ms±6.930	1004.155 ms±7.436	250.906 ms±5.931	321.850 ms±1.125	150.975 ms±1.987
	4.820±0.140	4.004±0.119		1.283±0.037	0.602±0.021
Havlak (1500)	3983.160 ms±3.491	3299.310 ms±2.599	2865.806 ms±6.263	1573.135 ms±3.163	1053.280 ms±2.459
	1.390±0.004	1.151±0.003		0.549±0.002	0.368±0.001
Json (100)	2049.190 ms±1.752	1698.275 ms±1.913	1058.888 ms±3.527	535.910 ms±0.845	226.130 ms±0.782
	1.935±0.008	1.604±0.007		0.506±0.002	0.214±0.001
List (1500)	606.875 ms±0.919	376.440 ms±0.841	507.136 ms±0.005	902.750 ms±1.960	454.665 ms±0.594
	1.197±0.002	0.742±0.002		1.780±0.005	0.897±0.001
Mandelbrot (500)	158.385 ms±0.807	149.995 ms±0.829	738.596 ms±0.004	145.680 ms±0.565	140.640 ms±0.249
	0.214±0.001	0.203±0.001		0.197±0.001	0.190±0.000
NBody (250000)	1041.260 ms±1.650	892.445 ms±1.251	1036.211 ms±0.039	279.940 ms±0.912	197.615 ms±0.684
	1.005±0.002	0.861±0.001		0.270±0.001	0.191±0.001
Permute (1000)	455.080 ms±0.877	470.890 ms±1.045	542.213 ms±0.034	237.210 ms±0.681	213.070 ms±0.709
	0.839±0.002	0.868±0.002		0.437±0.002	0.393±0.002
Queens (1000)	475.140 ms±0.873	384.660 ms±1.105	395.362 ms±0.015	260.710 ms±0.371	177.770 ms±1.316
	1.202±0.003	0.973±0.003		0.659±0.001	0.450±0.004
Richards (100)	2661.565 ms±1.867	2226.210 ms±1.904	1138.765 ms±0.174	1114.975 ms±1.029	882.705 ms±1.049
	2.337±0.002	1.955±0.002		0.979±0.001	0.775±0.001
Sieve (3000)	222.140 ms±1.614	235.605 ms±1.196	713.322 ms±2.111	218.760 ms±1.478	169.130 ms±0.903
	0.311±0.003	0.330±0.002		0.307±0.003	0.237±0.002
Storage (1000)	329.000 ms±1.114	318.210 ms±0.968	479.860 ms±1.794	304.785 ms±0.753	185.270 ms±0.871
	0.686±0.004	0.663±0.004		0.635±0.003	0.386±0.003
Towers (600)	910.670 ms±1.200	823.625 ms±1.814	441.156 ms±0.017	372.420 ms±2.437	275.885 ms±0.714
	2.064±0.003	1.867±0.005		0.844±0.008	0.625±0.002
Geometric mean	1.171±0.004	1.003±0.004		0.571±0.003	0.414±0.002

However, the performance impact of the latency mode, which disables inlining and splitting in the compiler, is much larger compared with the difference in frame rates measured as part of the UI benchmark shown in Figure 11.4a. In throughput mode, so with inlining and splitting, the macro benchmarks run more than twice as fast. *DeltaBlue*, a constraint solver benchmark, and *Havlak*, a loop recognition algorithm, are almost four times faster.

Another expected result demonstrated by these macro benchmarks is that GraalVM EE, which performs additional performance optimizations, is generally faster than GraalVM CE. *Richards*, an OS kernel simulation benchmark, on TruffleSqueak in throughput mode, for example, needs around 1115 ms on GraalVM CE and 883 ms on GraalVM EE. *Json*, a JSON string parsing benchmark, is more than twice as fast on GraalVM EE compared with GraalVM CE. The OpenSmalltalkVM is somewhere between TruffleSqueak on GraalVM EE in latency and throughput mode in terms of performance. For *Havlak*, it is close to the performance of GraalVM EE in latency mode but close to the throughput mode for *Richards*. In the *DeltaBlue* benchmark, the OpenSmalltalkVM also comes close to GraalVM EE in throughput mode and outperforms GraalVM CE in that mode.

From the nine remaining benchmarks, all of which are considered micro benchmarks, only *Queens*, an eight queens puzzle solver, and *Towers*, a solver for the Towers of Hanoi game, produce results similar to the macro benchmarks. All others deviate from this pattern. In *NBody*, a solar system simulation, the OpenSmalltalkVM is slower than TruffleSqueak on GraalVM EE in latency mode but still comparable to GraalVM CE. In four benchmarks, however, the OpenSmalltalkVM performs significantly worse than all TruffleSqueak configurations: 1) *Mandelbrot*, a fractal generator, 2) *Permute*, a benchmark generating permutations of an array, 3) *Sieve*, an implementation of the sieve of Eratosthenes algorithm for finding prime numbers, and 4) *Storage*, a benchmark building up a tree of arrays to exercise the GC.

The two remaining micro benchmarks — *Bounce* and *List* — show unexpected results: In *Bounce*, a benchmark simulating a ball bouncing in a box, TruffleSqueak on GraalVM EE in throughput mode performs worse than GraalVM CE. Since this is considered a performance issue, we reported it to the GraalVM team. For *List*, a benchmark creating and traversing lists recursively, the compilation statistics reveal that 387 method splits occurred when running on GraalVM CE in throughput mode. This in turn led to 148 successful compilations in total and run-times at around 903 ms. Although GraalVM CE in latency mode only compiled 39 compilations for the same benchmark, it performed with around 607 ms per iteration much better than in throughput mode. The high number of splits that led to significantly more compiled code is caused by TruffleSqueak reporting polymorphism to Truffle

in one of its nodes for message dispatching. That the run-time performance of the *List* benchmark is better when inlining and splitting is disabled is a result of this particular performance evaluation. Our continuous performance tracking infrastructure did not reveal this problem because it only runs the AWFY benchmarks against GraalVM CE in throughput mode. After further investigation, we found out that the *List* benchmark on TruffleSqueak is a worst case for the splitting heuristic of the Graal compiler. The reason for this is that the benchmark is not only highly recursive but also polymorphic in Squeak/Smalltalk due to the `isNil` method, which is implemented on `ProtoObject` and `UndefinedObject`. We reported our findings to the GraalVM team, which plans to improve the splitting heuristic in a future GraalVM release. In the past, similar splitting misbehavior that we observed with TruffleSqueak helped to detect a serious performance regression caused by a bug in Graal's splitting heuristic.

Overall, we conclude that TruffleSqueak in latency mode needs around 117.1 % of the time of the OpenSmalltalkVM and almost the same on GraalVM EE to run all benchmarks. With the throughput mode, performance is increased significantly so that TruffleSqueak only needs approximately 57.1 % of the time on GraalVM CE compared with the OpenSmalltalkVM. On GraalVM EE, it is with only around 41.4 % of the time even more than twice as fast. We believe that TruffleSqueak's performance advantages over the OpenSmalltalkVM are due to the significantly larger engineering efforts that went into GraalVM, the JVM, the JVM garbage collectors. As discussed in Section 11.2, however, TruffleSqueak generally requires significantly more CPU and memory resources to run compared with the OpenSmalltalkVM. While first experiments with GraalVM's Native Image technology suggest that AOT compilation can reduce CPU and memory footprint, we expect to see further improvements in both performance and footprint for GraalVM in general.

Appendix C.

Additional Screenshots

In this chapter, we show and describe additional screenshots of two polyglot notebooks as well as a `CallTargetBrowser` tool for GraalPython.

Polyglot Notebook Analyzing Object Layouts in TruffleSqueak Figure C.1 shows a polyglot notebook for analyzing TruffleSqueak's object layout optimizations (see Section 8.1) at run-time. It uses Smalltalk (green code cell) to enumerate all classes for pointers objects and to find the corresponding layout objects from its language implementation through VM introspection and interoperability with the host language. Finally, it extracts and exports a

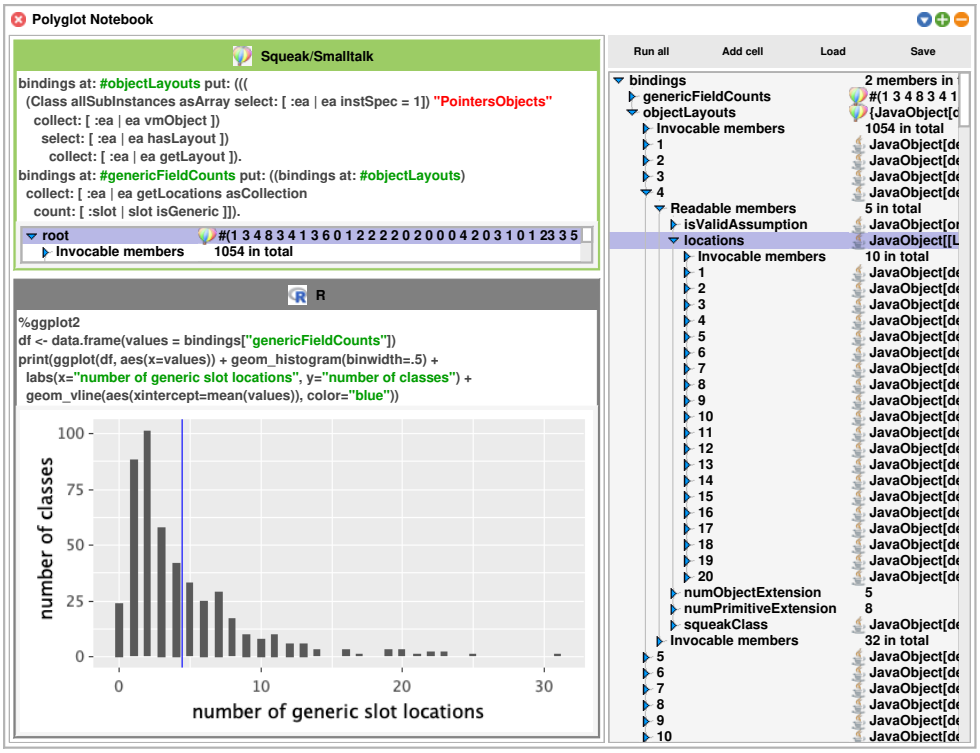


Figure C.1.: A polyglot notebook analyzing object layouts in TruffleSqueak.

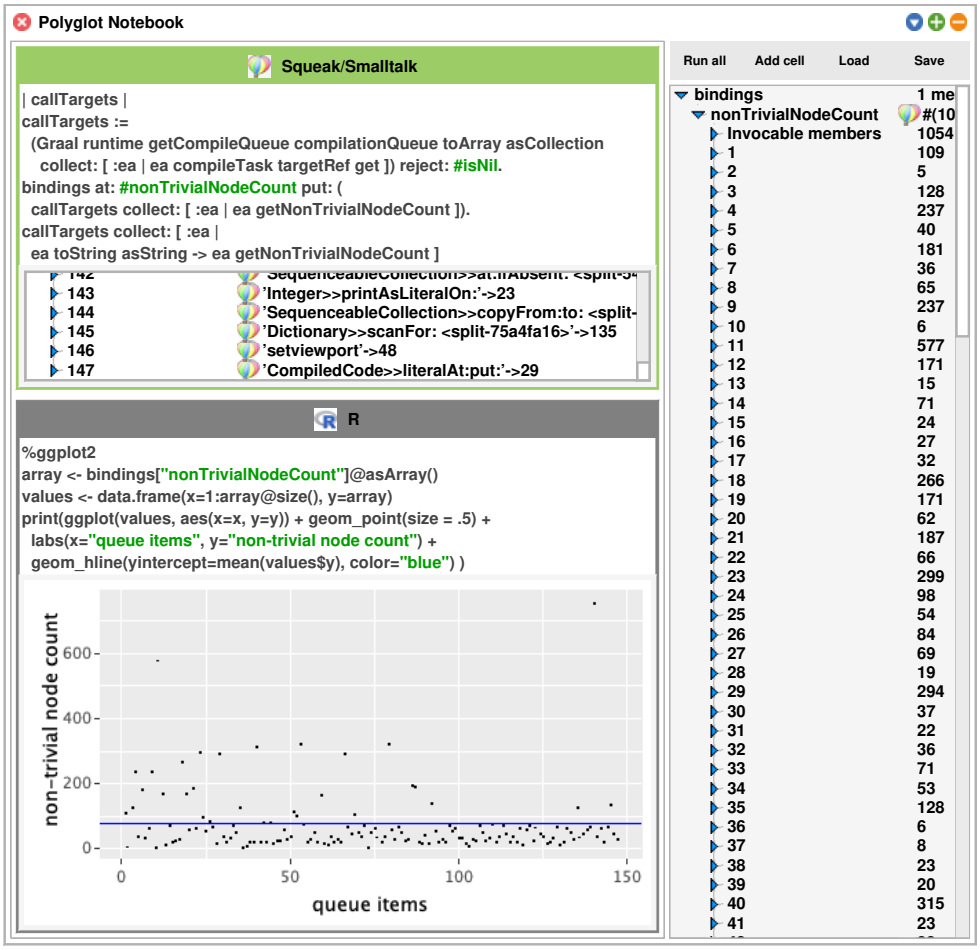


Figure C.2.: A polyglot notebook analyzing the Graal compilation queue.

list of generic field counts. In R (gray code cell), these counts are visualized with the `ggplot2` package. In this case, the histogram shows how many of these Smalltalk classes have how many generic slot locations in their object layouts. This type of analysis can help to verify that the optimization in the object model works as intended. We further used such run-time data to find appropriate sizes for optimizations in TruffleSqueak. The plot shows that the mean of generic locations is around four. In TruffleSqueak, we decided to use three inline `Object` fields to keep some room for improvements when needed. With this notebook, it is also straightforward to examine other aspects of the object layout optimization such as the number of primitive or uninitialized locations or the sizes of extension arrays.

Polyglot Notebook Analyzing the Graal Compilation Queue Figure C.2 shows a polyglot notebook for analyzing the items of the Graal compilation

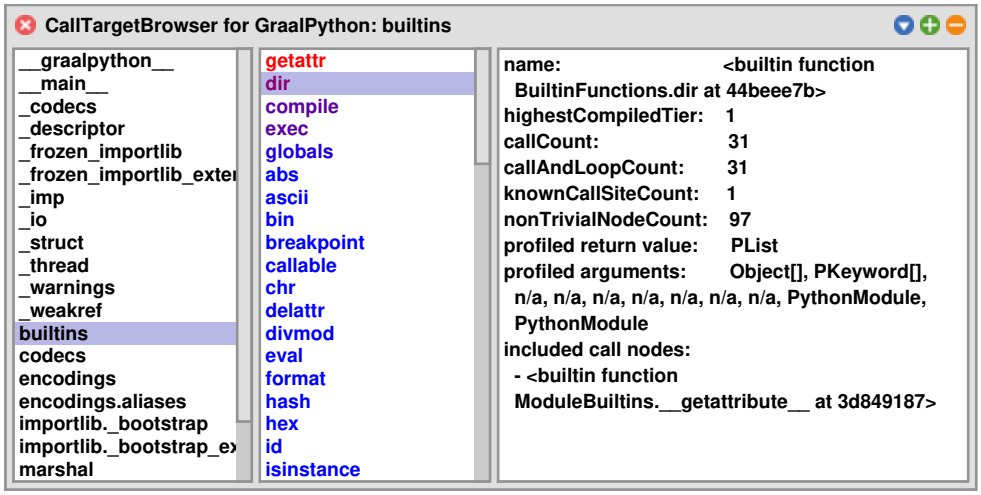


Figure C.3.: The `CallTargetBrowserPython` tool for GraalPython.

queue according to their `nonTrivialNodeCount` at run-time. This value indicates how complex the AST for a method is according to the corresponding language implementation. It is used for different purposes within the Graal compiler. Prior to GraalVM 20.2.0, for example, the count was used for inlining and splitting and is still used for other optimizations. The first code cell (green) uses Smalltalk to take a snapshot of the call targets that are currently in the Graal compilation queue through VM introspection and interoperability with the host language. It also extracts and exports a list of non-trivial node counts and outputs a list of the call targets and their corresponding node count. According to this list, many of the call targets in the queue represent Smalltalk methods and come thus from TruffleSqueak. The 146th call target, however, is attached to an R method that is called as part of FastR's `grid` package. This means that R code was executed before the last execution of the green code cell. In the gray code cell, the list of node counts is visualized with the `ggplot2` package from R. The plot shows that most of the 147 call targets that were snapshotted have a non-trivial node count below 100. However, two call targets have counts above 500 and are thus interesting candidates for further investigations. These can reveal problems in TruffleSqueak's language implementation or in the Graal compiler. A large non-trivial node count can, for example, indicate an important method that has been subject to a lot of inlining but also problems in how TruffleSqueak constructs ASTs or estimates the costs of specific nodes.

A `CallTargetBrowser` for GraalPython Figure C.3 shows a screenshot of a `CallTargetBrowserPython` tool in TruffleSqueak for GraalPython. It is used to inspect the Python `dir` built-in function at run-time. The tool is implemented

as a subclass of the `CallTargetBrowserRuby` shown in Figure 12.9 with five method overrides and 25 SLOC, providing the same set of features but for call targets from GraalPython. It is thus another example of how quickly tools can be built and adapted for specific purposes and language implementations in TruffleSqueak.

Publications

Journals

- Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. "Live Multi-language Development and Runtime Environments". In: *The Art, Science, and Engineering of Programming* 2.3 (Mar. 2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/8

Conferences

- Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. "Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA: Association for Computing Machinery, 2020, pages 1–17. ISBN: 978-1-4503-8178-9
- Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "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*. MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 14–26. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361024
- Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. "Language-Independent Development Environment Support for Dynamic Runtimes". In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 80–90. ISBN: 978-1-4503-6996-1. DOI: 10.1145/3359619.3359746
- Matthias Springer, Fabio Niephaus, Robert Hirschfeld, and Hidehiko Masuhara. "Matriona: Class Nesting with Parameterization in Squeak/Smalltalk". In: *Proceedings of the 15th International Conference on Modularity*. MODULARITY 2016. Málaga, Spain: Association for Computing Machinery, 2016, pages 118–129. ISBN: 978-1-4503-3995-7. DOI: 10.1145/2889443.2889457

Workshops

- Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "User-Defined Interface Mappings for the GraalVM". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 19–22. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3399577
- Jan Ehmueller, Alexander Riese, Hendrik Tjabben, Fabio Niephaus, and Robert Hirschfeld. "Polyglot Code Finder". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 106–112. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3397559
- Johannes Henning, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. "Toward Presizing and Pretransitioning Strategies for GraalPython". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 41–45. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3397564
- Johannes Henning, David Stangl, Fabio Niephaus, Bastian Kruck, and Robert Hirschfeld. "Hot Code Patching in CPython: Supporting Edit-and-Continue Debugging in CPython with Less Than 300 Lines of Code". In: *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '19. London, United Kingdom: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6862-9. DOI: 10.1145/3340670.3342424
- Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. "Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-Based Live Programming as a Use Case for Context-Oriented Programming". In: *Proceedings of the Workshop on Context-Oriented Programming*. COP '19. London, United Kingdom: Association for Computing Machinery, 2019, pages 17–23. ISBN: 978-1-4503-6863-6. DOI: 10.1145/3340671.3343358
- Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. "PolyJuS: A Squeak/Smalltalk-Based Polyglot Notebook System for the GraalVM". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328434

- Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. “Towards Polyglot Adapters for the GraalVM”. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› ’19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328458
- Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. “Efficient Implementation of Smalltalk Activation Records in Language Implementation Frameworks”. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› ’19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328440
- Tobias Pape, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. “Let Them Fail: Towards VM Built-in Behavior That Falls Back to the Program”. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› ’19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3338056
- Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. “GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework”. In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pages 30–35. ISBN: 978-1-4503-5804-0. DOI: 10.1145/3242947.3242948
- Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. “Squeak Makes a Good Python Debugger: Bringing Other Programming Languages Into Smalltalk’s Tools”. In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. ‹Programming› ’17. Brussels, Belgium: Association for Computing Machinery, 2017. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079402
- Fabio Niephaus. “Towards A Squeak/Smalltalk-Based Python IDE: An Interpreter-Level Integration of Python with Smalltalk”. In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. ‹Programming› ’17. Brussels, Belgium: Association for Computing Machinery, 2017. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079370
- Fabio Niephaus, Dale Henrichs, Marcel Taeumel, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. “SmalltalkCI: A Continuous Integration Framework for Smalltalk Projects”. In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST’16. Prague, Czech Republic: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991044

- Fabio Niephaus, Matthias Springer, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Call-Target-Specific Method Arguments". In: *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '15. Prague, Czech Republic: Association for Computing Machinery, 2015. ISBN: 978-1-4503-3657-4. DOI: 10.1145/2843915.2843919

Technical Reports

- Christian Adriano, Tobias Bleifuß, Lung-Pan Cheng, Kiarash Diba, Andreas Fricke, Andreas Grapentin, Lan Jiang, Robert Kovacs, Martin Krejca, Sankalita Mandal, Sebastian Marwecki, Christoph Matthies, Toni Mattis, Fabio Niephaus, Lukas Pirl, Francesco Quinzan, Stefan Ramson, Mina Rezaei, Julian Risch, Ralf Rothenberger, Thijs Roumen, Vladeta Stojanovic, and Johannes Wolf. *Technical report: Fall Retreat 2018*. Technical report 129. Hasso-Plattner-Institut, 2019. DOI: 10.25932/publishup-42753
- Jakob Reschke, Marcel Taeumel, Tobias Pape, Fabio Niephaus, and Robert Hirschfeld. *Towards Version Control in Object-Based Systems*. Technical report 121. Hasso-Plattner-Institut, 2018
- Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. *Squimera: A Live, Smalltalk-Based IDE for Dynamic Programming Languages*. Technical report 120. Hasso-Plattner-Institut, 2017. DOI: 10.25932/publishup-40338
- Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Robert Hirschfeld, and Carsten Witt. *ecoControl: Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern*. Technical report 93. Hasso-Plattner-Institut, 2015

Bibliography

[1] Christian Adriano, Tobias Bleifuß, Lung-Pan Cheng, Kiarash Diba, Andreas Fricke, Andreas Grapentin, Lan Jiang, Robert Kovacs, Martin Krejca, Sankalita Mandal, Sebastian Marwecki, Christoph Matthies, Toni Mattis, Fabio Niephaus, Lukas Pirl, Francesco Quinzan, Stefan Ramson, Mina Rezaei, Julian Risch, Ralf Rothenberger, Thijs Roumen, Vladeta Stojanovic, and Johannes Wolf. *Technical report: Fall Retreat 2018*. Technical report 129. Hasso-Plattner-Institut, 2019. DOI: 10.25932/publishup-42753.

[2] Amber contributors. *Amber Smalltalk*. 2021. URL: https://amber-lang.net (visited on 2021-08-17).

[3] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. "Programming Languages for Distributed Computing Systems". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pages 261–322. ISSN: 0360-0300. DOI: 10.1145/72551.72552.

[4] Gergö Barany. "Python Interpreter Performance Deconstructed". In: *Proceedings of the Workshop on Dynamic Languages and Applications*. Dyla'14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pages 1–9. ISBN: 978-1-4503-2916-3. DOI: 10.1145/2617548.2617552.

[5] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. "Approaches to Interpreter Composition". In: *Computer Languages, Systems and Structures*. Volume abs/1409.0757. Elsevier, Mar. 2015. DOI: http://dx.doi.org/10.1016/j.cl.2015.03.001.

[6] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. "Unipycation: A Case Study in Cross-Language Tracing". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 31–40. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542146.

[7] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. "Virtual Machine Warmup Blows Hot and Cold". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133876.

[8] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. "Pycket: A Tracing JIT for a Functional Language". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pages 22–34. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784740.

[9] David M. Beazley. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++". In: *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*. TCLTK'96. Monterey, California: USENIX Association, 1996, page 15.

[10] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. "Cython: The Best of Both Worlds". In: *Computing in Science Engineering* 13.2 (2011), pages 31–39. DOI: 10.1109/MCSE.2010.118.

[11] Nick Benton, Andrew Kennedy, and George Russell. "Compiling Standard ML to Java Bytecodes". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pages 129–140. ISBN: 1-58113-024-4. DOI: 10.1145/289423.289435.

[12] Clément Béra and Eliot Miranda. "A bytecode set for adaptive optimizations". In: *Proceedings of the 6th Edition of the International Workshop on Smalltalk Technologies*. IWST'14. Cambridge, England, 2014.

[13] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. "Sista: Saving Optimized Code in Snapshots for Fast Start-Up". In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: Association for Computing Machinery, 2017, pages 1–11. ISBN: 978-1-4503-5340-3. DOI: 10.1145/3132190.3132201.

[14] Brian Blount and Siddhartha Chatterjee. "An Evaluation of Java for Numerical Computing". In: *Computing in Object-Oriented Parallel Environments*. Edited by Denis Caromel, Rodney R. Oldehoeft, and Marydell Tholburn. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 35–46. ISBN: 978-3-540-49372-3.

[15] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. "Back to the Future in One Week – Implementing a Smalltalk VM in PyPy". In: *Self-Sustaining Systems: First Workshop, S3 2008 Pots-*

dam, Germany, May 15-16, 2008 Revised Selected Papers. Edited by Robert Hirschfeld and Kim Rose. Berlin, Heidelberg: Springer-Verlag, 2008, pages 123–139. ISBN: 978-3-540-89275-5. DOI: 10.1007/978-3-540-89275-5_7.

[16] Per Bothner. "Kawa: Compiling Dynamic Languages to the Java VM". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '98. New Orleans, Louisiana: USENIX Association, 1998, page 41.

[17] Benedict du Boulay. "POPLOG for Beginners: A Powerful Environment for Learning Programming". In: *Artificial Intelligence Programming Environments*. USA: John Wiley & Sons, Inc., 1987, pages 31–42. ISBN: 0-470-20989-5.

[18] Don Box and Chris Sells. *Essential .NET: The Common Language Runtime*. Addison-Wesley, 2002. ISBN: 0-201-73411-7.

[19] Experience in Bridging Keras for Python with Pharo. "Infante, Alejandro and Bergel, Alexandre". In: *Proceedings of the 13th Edition of the International Workshop on Smalltalk Technologies*. IWST '18. Cagliari, Italy, 2018.

[20] Thorsten Brunklaus and Leif Kornstaedt. *A Virtual Machine for Multi-Language Execution*. Submitted. Nov. 2002.

[21] P.A. Buhr and W.Y.R. Mok. "Advanced exception handling mechanisms". In: *IEEE Transactions on Software Engineering* 26.9 (2000), pages 820–836. DOI: 10.1109/32.877844.

[22] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. "Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices". In: *2009 Third International Conference on Sensor Technologies and Applications*. 2009, pages 117–125. DOI: 10.1109/SENSORCOMM.2009.27.

[23] C. Chambers, D. Ungar, and E. Lee. "An Efficient Implementation of Self a Dynamically-Typed Object-Oriented Language Based on Prototypes". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '89. New Orleans, Louisiana, USA: Association for Computing Machinery, 1989, pages 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74884.

[24] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. "Fully Reflective Execution Environments: Virtual Machines for More Flexible Software". In: *IEEE Transactions on Software Engineering* 45.9 (2019), pages 858–876. DOI: 10.1109/TSE.2018.2812715.

[25] Jessie Y. C. Chen and Jennifer E. Thropp. "Review of Low Frame Rate Effects on Human Performance". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 37.6 (2007), pages 1063–1076. DOI: 10.1109/TSMCA.2007.904779.

[26] Yaofei Chen, R. Dios, A. Mili, Lan Wu, and Kefei Wang. "An empirical study of programming language trends". In: *IEEE Software* 22.3 (2005), pages 72–79. DOI: 10.1109/MS.2005.55.

[27] David Chisnall. "The Challenge of Cross-Language Interoperability". In: *Commun. ACM* 56.12 (Dec. 2013), pages 50–56. ISSN: 0001-0782. DOI: 10.1145/2534706.2534719.

[28] Fred Chow. "Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers". In: *Queue* 11.10 (Oct. 2013), pages 30–37. ISSN: 1542-7730. DOI: 10.1145/2542661.2544374.

[29] GraalVM Community. *GraalVM CE vm-21.2.0 release tag*. 2021. URL: https://git.io/JuDzb (visited on 2021-09-13).

[30] OpenJDK Community. *JEP 243: Java-Level JVM Compiler Interface*. 2014. URL: https://openjdk.java.net/jeps/243 (visited on 2021-02-05).

[31] Marcus Crestani. "Foreign-Function Interfaces for Garbage-Collected Programming Languages". In: *Proceedings of the Workshop on Scheme and Functional Programming 2008*. 2008.

[32] Andrew Davison, Michael Hines, and Eilif Muller. "Trends in programming languages for neuroscience simulations". In: *Frontiers in Neuroscience* 3 (2009), page 36. ISSN: 1662-453X. DOI: 10.3389/neuro.01.036.2009.

[33] J. Des Rivières and J. Wiegand. "Eclipse: A Platform for Integrating Development Tools". In: *IBM Syst. J.* 43.2 (Apr. 2004), pages 371–383. ISSN: 0018-8670. DOI: 10.1147/sj.432.0371.

[34] Lukas Diekmann and Laurence Tratt. "Default Disambiguation for Online Parsers". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 88–99. ISBN: 978-1-4503-6981-7. DOI: 10.1145/3357766.3359530.

[35] Lukas Diekmann and Laurence Tratt. "Eco: A Language Composition Editor". In: *Software Language Engineering*. Edited by Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju. Cham: Springer International Publishing, 2014, pages 82–101. ISBN: 978-3-319-11245-9.

[36] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 1–10. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542143.

[37] Jan Ehmueller, Alexander Riese, Hendrik Tjabben, Fabio Niephaus, and Robert Hirschfeld. "Polyglot Code Finder". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 106–112. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3397559.

[38] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. "The State of the Art in Language Workbenches". In: *Software Language Engineering*. Edited by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pages 197–217. ISBN: 978-3-319-02654-1.

[39] Moritz Eysholdt and Heiko Behrens. "Xtext: Implement Your Language Faster than the Quick and Dirty Way". In: *Proceedings of the ACM International Conference Companion on Object-Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pages 307–309. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869625.

[40] feenk GmbH, Switzerland. *Sparta*. 2021. URL: https://github.com/feenkcom/sparta (visited on 2021-07-11).

[41] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. "How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST'16. Prague, Czech Republic: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.

[42] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. "How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain".

In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST '16. Prague, Czech Republic: ACM, 2016, 21:1–21:10. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991062.

[43] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. "A Dynamically Configurable, Multi-Language Execution Platform". In: *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*. EW 8. Sintra, Portugal: Association for Computing Machinery, 1998, pages 175–181. ISBN: 978-1-4503-7317-3. DOI: 10.1145/319195.319222.

[44] B. Foote and R. E. Johnson. "Reflective Facilities in Smalltalk-80". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '89. New Orleans, Louisiana, USA: Association for Computing Machinery, 1989, pages 327–335. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74911.

[45] Mathieu Fourment and Michael R. Gillings. "A comparison of common programming languages used in bioinformatics". In: *BMC Bioinformatics* 9.1 (2008), page 82. DOI: 10.1186/1471-2105-9-82.

[46] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: https://martinfowler.com/articles/languageWorkbench.html (visited on 2021-08-01).

[47] Bert Freudenberg, Dan H.H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser". In: *SIGPLAN Not.* 50.2 (Oct. 2014), pages 57–66. ISSN: 0362-1340. DOI: 10.1145/2775052.2661100.

[48] Yoshihiko Futamura. "Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler". In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pages 381–391. ISSN: 1388-3690. DOI: 10.1023/A:1010095604496.

[49] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer-Verlag, 2010. DOI: 10.1007/978-1-84882-914-5.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[51] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. second. Prentice-Hall, Inc., Sept. 2003. ISBN: 81-203-2242-8.

[52] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0-201-11372-4.

[53] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6.

[54] Google. *Protocol Buffers*. 2021. URL: https://developers.google.com/protocol-buffers (visited on 2021-08-05).

[55] Google. *Skia*. 2021. URL: https://skia.org/ (visited on 2021-07-11).

[56] Google Research. *Colaboratory*. 2021. URL: http://colab.research.google.com (visited on 2021-07-03).

[57] Alfred Gray. *Interprocess Communication in Linux*. Prentice Hall Professional Technical Reference, 2002. ISBN: 0-13-046042-7.

[58] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. "High-Performance Cross-Language Interoperability in a Multi-Language Runtime". In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pages 78–90. ISBN: 978-1-4503-3690-1. DOI: 10.1145/2816707.2816714.

[59] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pages 185–200. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363.

[60] Jennifer Hamilton. "Language Integration in the Common Language Runtime". In: *SIGPLAN Not.* 38.2 (Feb. 2003), pages 19–28. ISSN: 0362-1340. DOI: 10.1145/772970.772973.

[61] Jeff Hardy. "The Dynamic Language Runtime and the Iron Languages". In: *The Architecture of Open Source Applications* (2008).

[62] Michael Haupt, Michael Perscheid, and Robert Hirschfeld. "Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: Association for Computing Machinery, 2011, pages 1282–1289. ISBN: 978-1-4503-0113-8. DOI: 10.1145/1982185.1982464.

[63] Johannes Henning, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. "Toward Presizing and Pretransitioning Strategies for GraalPython". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 41–45. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3397564.

[64] Johannes Henning, David Stangl, Fabio Niephaus, Bastian Kruck, and Robert Hirschfeld. "Hot Code Patching in CPython: Supporting Edit-and-Continue Debugging in CPython with Less Than 300 Lines of Code". In: *Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '19. London, United Kingdom: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6862-9. DOI: 10.1145/3340670.3342424.

[65] Eva-Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, Marcel Taeumel, Robert Hirschfeld, and Carsten Witt. *ecoControl: Entwurf und Implementierung einer Software zur Optimierung heterogener Energiesysteme in Mehrfamilienhäusern*. Technical report 93. Hasso-Plattner-Institut, 2015.

[66] Marcel Hlopko, Jan Kurš, Jan Vraný, and Claus Gittinger. "On the Integration of Smalltalk and Java: Practical Experience with STX:LIBJAVA". In: *Proceedings of the International Workshop on Smalltalk Technologies*. IWST '12. Ghent, Belgium: Association for Computing Machinery, 2012. ISBN: 978-1-4503-1897-6. DOI: 10.1145/2448963.2448968.

[67] Reid Holmes and David Notkin. "Enhancing Static Source Code Search with Dynamic Data". In: *Proceedings of 2010 ICSE Workshop on Search-Driven Development: Users, Infrastructure, Tools and Evaluation*. SUITE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pages 13–16. ISBN: 978-1-60558-962-6. DOI: 10.1145/1809175.1809179.

[68] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches". In: *ECOOP'91 European Conference on Object-Oriented Programming*. Edited by Pierre America. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pages 21–38. ISBN: 978-3-540-47537-8.

[69] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. "A Domain-Specific Language for Building Self-Optimizing AST Interpreters". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. Västerås, Sweden: Association for Computing Machinery,

2014, pages 123–132. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658776.

[70] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '97. Atlanta, Georgia, USA: Association for Computing Machinery, 1997, pages 318–326. ISBN: 0-89791-908-4. DOI: 10.1145/263698.263754.

[71] Daniel Ingalls. "The Evolution of Smalltalk: From Smalltalk-72 through Squeak". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386335.

[72] Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. "A World of Active Objects for Work and Play: The First Ten Years of Lively". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 238–249. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986029.

[73] Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. "Two decades of live coding and debugging of virtual machines through simulation". In: *Software: Practice and Experience* 50.9 (2020), pages 1629–1650. DOI: https://doi.org/10.1002/spe.2841. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2841.

[74] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The Lively Kernel A Self-supporting System on a Web Page". In: *Self-Sustaining Systems*. Edited by Robert Hirschfeld and Kim Rose. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 31–50. ISBN: 978-3-540-89275-5.

[75] IPython-contrib Developers. *Variable Inspector*. 2021. URL: https://git.io/JC0XC (visited on 2021-07-14).

[76] David Jackson and Gary Clynch. "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pages 154–160. DOI: 10.1109/UCC-Companion.2018.00050.

[77] JetBrains. *IntelliJ IDEA*. 2021. URL: https://www.jetbrains.com/idea/ (visited on 2021-08-10).

[78] JetBrains. *Meta Programming System*. 2021. URL: https://www.jetbrains.com/mps/ (visited on 2021-08-01).

[79] Steven G. Johnson. *PyCall.jl: Calling Python functions from the Julia language*. 2021. URL: https://github.com/JuliaPy/PyCall.jl (visited on 2021-04-07).

[80] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011. ISBN: 1-4200-8279-5.

[81] Tomas Kalibera and Richard Jones. *Quantifying Performance Changes with Effect Size Confidence Intervals*. Technical Report 4–12. University of Kent, June 2012, page 55.

[82] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/-Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pages 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[83] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The IDE Portability Problem and Its Solution in Monto". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 152–162. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997368.

[84] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". In: *ECOOP 2001 — Object-Oriented Programming*. Edited by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 327–354. ISBN: 978-3-540-45337-6.

[85] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming*. Edited by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pages 220–242. ISBN: 978-3-540-69127-3.

[86] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-11158-6.

[87] Paul King. "A History of the Groovy Programming Language". In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386326.

[88] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. "Jupyter Notebooks ? a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Edited by Fernando Loizides and Birgit Scmidt. IOS Press, 2016, pages 87–90.

[89] D. E. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (Jan. 1984), pages 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97.

[90] Charles W. Krueger. "Software Reuse". In: *ACM Comput. Surv.* 24.2 (June 1992), pages 131–183. ISSN: 0360-0300. DOI: 10.1145/130844.130856.

[91] David Alex Lamb. "IDL: Sharing Intermediate Representations". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pages 297–318. ISSN: 0164-0925. DOI: 10.1145/24039.24040.

[92] Craig Latta. *Caffeine – Livecode the Web!* 2021. URL: https://caffeine.js.org (visited on 2021-08-05).

[93] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, page 75. ISBN: 0-7695-2102-9.

[94] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. "The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2020, pages 1–11. DOI: 10.1109/VL/HCC50065.2020.9127201.

[95] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. "Debug All Your Code: Portable Mixed-Environment Debugging". In: *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pages 207–226. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640105.

[96] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. "Designing a Live Development Experience for Web-Components". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience*. PX/17.2. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pages 28–35. ISBN: 978-1-4503-5522-3. DOI: 10.1145/3167109.

[97] P.K. Linos. "PolyCARE: a tool for re-engineering multi-language program integrations". In: *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95*. 1995, pages 338–341. DOI: 10.1109/ICECCS.1995.479355.

[98] Stephan Lutz. *Squeak Graphics OpenGL*. 2021. URL: https://github.com/hpi-swa-lab/squeak-graphics-opengl (visited on 2021-07-11).

[99] John Maloney. *Morphic: The Self User Interface Framework*. Sun Microsystems Laboratories. 1995.

[100] John H. Maloney and Randall B. Smith. "Directness and Liveness in the Morphic User Interface Construction Environment". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1995, pages 21–28. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215636.

[101] Ami Marowka. "Python accelerators for high-performance computing". In: *The Journal of Supercomputing* 74.4 (2018), pages 1449–1460. DOI: 10.1007/s11227-017-2213-5.

[102] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. "Cross-Language Compiler Benchmarking: Are We Fast Yet?" In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 120–131. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989232.

[103] Philip Mayer and Andreas Schroeder. "Cross-Language Code Analysis and Refactoring". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. 2012, pages 94–103. DOI: 10.1109/SCAM.2012.11.

[104] Robert McGill, John W. Tukey, and Wayne A. Larsen. "Variations of Box Plots". In: *The American Statistician* 32.1 (1978), pages 12–16. ISSN: 00031305. DOI: 10.2307/2683468.

[105] Wes McKinney. "pandas: a Foundational Python Library for Data Analysis and Statistics". In: *Python High Performance Science Computer* 14.9 (Jan. 2011), pages 1–9.

[106] Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell. "Experiences with Building Distributed Debuggers". In: *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. SPDT '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pages 70–79. ISBN: 0-89791-846-0. DOI: 10.1145/238020.238043.

[107] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pages 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[108] Leo A. Meyerovich and Ariel S. Rabkin. "Empirical Analysis of Programming Language Adoption". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 1–18. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509515.

[109] Microsoft. *.NET Interactive*. 2021. URL: https://github.com/dotnet/interactive (visited on 2021-08-01).

[110] Microsoft. *Debug Adapter Protocol*. 2021. URL: https://microsoft.github.io/debug-adapter-protocol/ (visited on 2021-08-10).

[111] Microsoft. *Language Server Protocol*. 2021. URL: https://microsoft.github.io/language-server-protocol/ (visited on 2021-08-10).

[112] Microsoft. *Visual Studio*. 2021. URL: https://visualstudio.microsoft.com (visited on 2021-08-02).

[113] Microsoft. *Visual Studio Code*. 2021. URL: https://code.visualstudio.com (visited on 2021-08-03).

[114] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. "Supporting On-Stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 1–13. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361030.

[115] Kenta Murata. *PyCall: Calling Python functions from the Ruby language*. 2021. URL: https://github.com/JuliaPy/PyCall.jl (visited on 2021-04-07).

[116] Netflix. *Polynote – The polyglot Scala notebook*. 2021. URL: https://polynote.org (visited on 2021-08-01).

[117] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. 1st. Springer-Verlag Berlin Heidelberg, 1999. DOI: 10.1007/978-3-662-03811-6.

[118] Fabio Niephaus. "Towards A Squeak/Smalltalk-Based Python IDE: An Interpreter-Level Integration of Python with Smalltalk". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. ‹Programming› '17. Brussels, Belgium: Association for Computing Machinery, 2017. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079370.

[119] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework". In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pages 30–35. ISBN: 978-1-4503-5804-0. DOI: 10.1145/3242947.3242948.

[120] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "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*. MPLR 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 14–26. ISBN: 978-1-4503-6977-0. DOI: 10.1145/3357390.3361024.

[121] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. *Squimera: A Live, Smalltalk-Based IDE for Dynamic Programming Languages*. Technical report 120. Hasso-Plattner-Institut, 2017. DOI: 10.25932/publishup-40338.

[122] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "Towards Polyglot Adapters for the GraalVM". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328458.

[123] Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Efficient Implementation of Smalltalk Activation Records in Language Implementation Frameworks". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328440.

[124] Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Squeak Makes a Good Python Debugger: Bringing Other Programming Languages Into Smalltalk's Tools". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming*. ‹Programming› '17. Brussels, Belgium: Association for Computing Machinery, 2017. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079402.

[125] Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. "Live Multi-language Development and Runtime Environments". In: *The Art, Science, and Engineering of Programming* 2.3 (Mar. 2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/8.

[126] Fabio Niephaus, Dale Henrichs, Marcel Taeumel, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. "SmalltalkCI: A Continuous Integration Framework for Smalltalk Projects". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies*. IWST'16. Prague, Czech Republic: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4524-8. DOI: 10.1145/2991041.2991044.

[127] Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. "PolyJuS: A Squeak/Smalltalk-Based Polyglot Notebook System for the GraalVM". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328434.

[128] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. "Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA: Association for Computing Machinery, 2020, pages 1–17. ISBN: 978-1-4503-8178-9.

[129] Fabio Niephaus, Matthias Springer, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. "Call-Target-Specific Method Arguments". In: *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '15. Prague, Czech Republic: Association for Computing Machinery, 2015. ISBN: 978-1-4503-3657-4. DOI: 10.1145/2843915.2843919.

[130] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. 1st. Pragmatic Bookshelf, 2011. ISBN: 978-1-934356-65-4.

[131] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Document Revision 2.0. OMG, July 1995.

[132] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An Overview of the Scala Programming Language*. Technical report. 2004.

[133] Travis E. Oliphant. "Python for Scientific Computing". In: *Computing in Science Engineering* 9.3 (2007), pages 10–20. DOI: 10.1109/MCSE.2007.58.

[134] OpenJDK Community. *Project Loom Wiki*. 2021. URL: https://wiki.openjdk.java.net/display/loom/Main (visited on 2021-07-28).

[135] Oracle. *Java Native Interface Specification*. 2020. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html (visited on 2021-04-07).

[136] Oracle Labs. *GraalVM Security Guide*. 2021. URL: https://www.graalvm.org/security-guide/ (visited on 2021-06-29).

[137] Oracle Labs. *GraalVM VS Code Extensions*. 2021. URL: https://github.com/graalvm/vscode-extensions (visited on 2021-07-05).

[138] Oracle Labs. *oracle/graal Repository: GraalVM SDK*. 2021. URL: https://git.io/JuDoo (visited on 2021-09-13).

[139] Oracle Labs. *oracle/graal Repository: InteropLibrary.java*. 2021. URL: https://git.io/JzZpY (visited on 2021-09-20).

[140] Oracle Labs. *oracle/graal Repository: Safepoints.md*. 2021. URL: https://git.io/JuDr5 (visited on 2021-09-13).

[141] Oracle Labs. *oracle/graal Repository: Splitting.md*. 2021. URL: https://git.io/JuDVM (visited on 2021-09-13).

[142] Oracle Labs. *oracle/graal Repository: Target Type Mappings*. 2021. URL: https://git.io/JB2HJ (visited on 2021-07-30).

[143] Oracle Labs. *oracle/graal Repository: TruffleLanguage.Env*. 2021. URL: https://git.io/JuD2n (visited on 2021-09-13).

[144] Oracle Labs. *oracle/graal Repository: TruffleObject*. 2021. URL: https://git.io/JuDan (visited on 2021-09-13).

[145] J. Pallas and D. Ungar. "Multiprocessor Smalltalk: A Case Study of a Multiprocessor-Based Programming Environment". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pages 268–277. ISBN: 0-89791-269-1. DOI: 10.1145/53990.54017.

[146] Tobias Pape, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. "Let Them Fail: Towards VM Built-in Behavior That Falls Back to the Program". In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '19. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3338056.

[147] Linda Dailey Paulson. "Developers shift to dynamic programming languages". In: *Computer* 40.2 (2007), pages 12–15. DOI: 10.1109/MC.2007.53.

[148] Bo Peng, Gao Wang, Jun Ma, Man Chong Leong, Chris Wakefield, James Melott, Yulun Chiu, Di Du, and John N Weinstein. "SoS Notebook: an interactive multi-language data analysis environment". In: *Bioinformatics* 34.21 (May 2018), pages 3768–3770. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty405. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/21/3768/26146920/bty405.pdf.

[149] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. "Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pages 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031.

[150] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0-262-16209-1.

[151] L. Prechelt. "An empirical comparison of seven programming languages". In: *Computer* 33.10 (2000), pages 23–29. DOI: 10.1109/2.876288.

[152] Project Jupyter. *Jupyter kernels*. 2021. URL: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels (visited on 2021-07-03).

[153] Python Software Foundation. *Extending and Embedding the Python Interpreter*. 2021. URL: https://docs.python.org/3/extending/index.html (visited on 2021-06-04).

[154] Patrick Rein, Robert Hirschfeld, and Marcel Taeumel. "Gramada: Immediacy in Programming Language Development". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 165–179. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986022.

[155] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. "Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-Based Live Programming as a Use Case for Context-Oriented Programming". In: *Proceedings of the Workshop on Context-Oriented Programming*. COP '19. London, United Kingdom: Association for Computing Machinery, 2019, pages 17–23. ISBN: 978-1-4503-6863-6. DOI: 10.1145/3340671.3343358.

[156] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. "Exploratory and Live, Programming and Coding". In: *The Art, Science, and Engineering of Programming* 3.1 (July 2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2019/3/1.

[157] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. "Language Boxes". In: *Software Language Engineering*. Edited by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 274–293. ISBN: 978-3-642-12107-4.

[158] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. "Embedding Languages without Breaking Tools". In: *ECOOP 2010 – Object-Oriented Programming*. Edited by Theo D'Hondt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 380–404. ISBN: 978-3-642-14107-2.

[159] John Reppy and Chunyan Song. "Application-Specific Foreign-Interface Generation". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pages 49–58. ISBN: 1-59593-237-2. DOI: 10.1145/1173706.1173714.

[160] Jakob Reschke, Marcel Taeumel, Tobias Pape, Fabio Niephaus, and Robert Hirschfeld. *Towards Version Control in Object-Based Systems*. Technical report 121. Hasso-Plattner-Institut, 2018.

[161] John C. Reynolds. "The Discoveries of Continuations". In: *Lisp Symb. Comput.* 6.3–4 (Nov. 1993), pages 233–248. ISSN: 0892-4635. DOI: 10.1007/BF01019459.

[162] Nicolas Riesco and contributors. *IJavascript*. 2021. URL: https://github.com/n-riesco/ijavascript (visited on 2021-07-14).

[163] Alexander Riese, Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. "User-Defined Interface Mappings for the GraalVM". In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*. ‹Programming› '20. Porto, Portugal: Association for Computing Machinery, 2020, pages 19–22. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3399577.

[164] Armin Rigo and Maciej Fijalkowski. *CFFI Documentation*. Technical report Release 1.14.6. 2021.

[165] Armin Rigo and Samuele Pedroni. "PyPy's approach to virtual machine construction". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA '06. Portland, USA: ACM, 2006, pages 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753.

[166] David Rothlisberger, Marcel Harry, Alex Villazon, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. "Augmenting static source views in IDEs with dynamic metrics". In: *2009 IEEE International Conference on Software Maintenance*. 2009, pages 253–262. DOI: 10.1109/ICSM.2009.5306302.

[167] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. "Exploiting Runtime Information in the IDE". In: *2008 16th IEEE International Conference on Program Comprehension*. 2008, pages 63–72. DOI: 10.1109/ICPC.2008.32.

[168] rpy2 Developers. *rpy2 - R in Python*. 2021. URL: https://rpy2.github.io (visited on 2021-08-16).

[169] Ruby Community. *Module: Fiddle (Ruby 3.0.2)*. 2021. URL: https://docs.ruby-lang.org/en/3.0.0/Fiddle.html (visited on 2021-08-16).

[170] Ruby Community. *To Ruby From C and C++*. 2021. URL: https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-c-and-cpp/ (visited on 2021-08-16).

[171] J. E. Sammet. "An Overview of Programming Languages for Specialized Application Areas". In: *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*. AFIPS '72 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1971, pages 299–311. ISBN: 978-1-4503-7909-0. DOI: 10.1145/1478873.1478912.

[172] D. W. Sandberg. "Smalltalk and Exploratory Programming". In: *SIGPLAN Not.* 23.10 (Oct. 1988), pages 85–92. ISSN: 0362-1340. DOI: 10.1145/51607.51614.

[173] Amin Shali and William R. Cook. "Hybrid Partial Evaluation". In: *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pages 375–390. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048098.

[174] M. Shaw. "Abstraction Techniques in Modern Programming Languages". In: *IEEE Softw.* 1.4 (Oct. 1984), pages 10–26. ISSN: 0740-7459. DOI: 10.1109/MS.1984.229453.

[175] Beau Sheil. "Datamation®: Power Tools for Programmers". In: *Readings in Artificial Intelligence and Software Engineering*. Edited by Charles Rich and Richard C. Waters. Morgan Kaufmann, 1986, pages 573–580. ISBN: 978-0-934613-12-5. DOI: https://doi.org/10.1016/B978-0-934613-12-5.50048-3.

[176] Tomomi Shimazaki, Masatomo Hashimoto, and Toshiyuki Maeda. "Developing a High-Performance Quantum Chemistry Program with a Dynamic Scripting Language". In: *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. SE-HPCCSE '15. Austin, Texas: Association for Computing Machinery, 2015, pages 9–15. ISBN: 978-1-4503-4012-0. DOI: 10.1145/2830168.2830170.

[177] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 3rd. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-69497-2.

[178] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation". In: *Facebook White Paper* 5.8 (2007).

[179] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1-55860-910-5.

[180] Robert Smith, Aaron Sloman, and John Gibson. "POPLOG's Two-level virtual machine support for interactive languages". In: *Research Directions in Cognitive Science Volume 5: Artificial Intelligence*. Edited by D. Sleeman and N. Bernsen. Lawrence Erlbaum Associates, 1992, pages 203–231.

[181] Software Architecture Group, Hasso Plattner Institute. *IPolyglot patch for the Node.js Evaluation Loop (NEL) module*. 2021. URL: https://git.io/JC0WE (visited on 2021-07-14).

[182] Software Architecture Group, Hasso Plattner Institute. *IPolyglot: A polyglot kernel for Jupyter notebooks based on GraalVM*. 2021. URL: https://github.com/hpi-swa/ipolyglot (visited on 2021-07-14).

[183] Software Architecture Group, Hasso Plattner Institute. *TruffleSqueak*. 2021. URL: https://github.com/hpi-swa/trufflesqueak/ (visited on 2021-07-05).

[184] Matthias Springer. *Inter-language Collaboration in an Object-oriented Virtual Machine*. 2016. arXiv: 1606.03644 [cs.PL].

[185] Matthias Springer, Fabio Niephaus, Robert Hirschfeld, and Hidehiko Masuhara. "Matriona: Class Nesting with Parameterization in Squeak/Smalltalk". In: *Proceedings of the 15th International Conference on Modularity*. MODULARITY 2016. Málaga, Spain: Association for Computing Machinery, 2016, pages 118–129. ISBN: 978-1-4503-3995-7. DOI: 10.1145/2889443.2889457.

[186] Squeak/Smalltalk Community. *Block / Brick (Pharo)*. 2021. URL: https://wiki.squeak.org/squeak/3806 (visited on 2021-07-11).

[187] Squeak/Smalltalk Community. *Method Finder*. 2019. URL: https://wiki.squeak.org/squeak/1916 (visited on 2021-07-07).

[188] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. "Partial Escape Analysis and Scalar Replacement for Java". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14. Orlando, FL, USA: Association for Computing Machinery, 2014, pages 165–174. ISBN: 978-1-4503-2670-4. DOI: 10.1145/2544137.2544157.

[189] Richard M. Stallman. "EMACS the Extensible, Customizable Self-Documenting Display Editor". In: *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. Portland, Oregon, USA: Association for Computing Machinery, 1981, pages 147–156. ISBN: 0-89791-050-8. DOI: 10.1145/800209.806466.

[190] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. "Inlining Java Native Calls at Runtime". In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. VEE '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pages 121–131. ISBN: 1-59593-047-7. DOI: 10.1145/1064979.1064997.

[191] Sam Stephenson and Josh Peek. *ExecJS*. 2021. URL: https://github.com/rails/execjs (visited on 2021-08-16).

[192] Daniel Stolpe, Tim Felgentreff, Christian Humer, Fabio Niephaus, and Robert Hirschfeld. "Language-Independent Development Environment Support for Dynamic Runtimes". In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2019. Athens, Greece: Association for Computing Machinery, 2019, pages 80–90. ISBN: 978-1-4503-6996-1. DOI: 10.1145/3359619.3359746.

[193] Dennis Strein and Hans Kratz. "Design and Implementation of a high-level multi-language .NET Debugger". In: *Proceedings of the .NET Technologies 2005 conference*. 2005, pages 57–64. ISBN: 80-86943-01-1.

[194] Gregory T. Sullivan. "Dynamic Partial Evaluation". In: *Programs as Data Objects*. Edited by Olivier Danvy and Andrzej Filinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 238–256. ISBN: 978-3-540-44978-2.

[195] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. "Efficient Dynamic Analysis for Node.Js". In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: Association for Computing Machinery, 2018, pages 196–206. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179527.

[196] Marcel Taeumel. "Data-driven Tool Construction in Exploratory Programming Environments". PhD thesis. University of Potsdam, Digital Engineering Faculty, Hasso Plattner Institute, Feb. 2020. DOI: 10.25932/publishup-44428.

[197] Marcel Taeumel and Robert Hirschfeld. "Evolving User Interfaces From Within Self-Supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. PX/16. Rome, Italy: Association for Computing Machinery, 2016, pages 43–59. ISBN: 978-1-4503-4776-1. DOI: 10.1145/2984380.2984386.

[198] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. "Partial Behavioral Reflection: Spatial and Temporal Selection of Reification". In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. OOPSLA '03. Anaheim, California, USA: Association for Computing Machinery, 2003, pages 27–46. ISBN: 1-58113-712-5. DOI: 10.1145/949305.949309.

[199] TensorFlow JVM Special Interest Group. *TensorFlow for Java*. 2021. URL: https://github.com/tensorflow/java (visited on 2021-08-16).

[200] The Khronos Group. *OpenGL*. 2021. URL: https://www.opengl.org (visited on 2021-07-11).

[201] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. "Languages as Libraries". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pages 132–141. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993514.

[202] Topaz Developers. *Topaz*. 2017. URL: https://github.com/topazproject/topaz (visited on 2021-07-15).

[203] J. Trenouth. "A Survey of Exploratory Software Development". In: *The Computer Journal* 34.2 (Jan. 1991), pages 153–163. ISSN: 0010-4620. DOI: 10.1093/comjnl/34.2.153. eprint: https://academic.oup.com/comjnl/article-pdf/34/2/153/1400604/340153.pdf.

[204] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. "How Do Programmers Ask and Answer Questions on the Web? (NIER Track)". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pages 804–807. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985907.

[205] John W Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977. ISBN: 978-0-201-07616-5.

[206] Two Sigma Open Source, LLC. *BakerX*. 2018. URL: http://beakerx.com (visited on 2021-07-03).

[207] David Ungar and Randall B. Smith. "Self". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 9–1–9–50. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238853.

[208] David Ungar and Randall B. Smith. "Self: The Power of Simplicity". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '87. Orlando, Florida, USA: Association for Computing Machinery, 1987, pages 227–242. ISBN: 0-89791-247-0. DOI: 10.1145/38765.38828.

[209] David Ungar, Adam Spitz, and Alex Ausch. "Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment". In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pages 11–20. ISBN: 1-59593-193-7. DOI: 10.1145/1094855.1094865.

[210] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. "Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools". In: *The Art, Science, and Engineering of Programming* 2.3 (Mar. 2018). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/14.

[211] Jan Vraný and Michal Píše. "Multilanguage Debugger Architecture". In: *SOFSEM 2010: Theory and Practice of Computer Science*. Edited by Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 731–742. ISBN: 978-3-642-11266-9.

[212] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. "Modular Semantic Actions". In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 108–119. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989231.

[213] Peter Wegner. "Interoperability". In: *ACM Comput. Surv.* 28.1 (Mar. 1996), pages 285–287. ISSN: 0360-0300. DOI: 10.1145/234313.234424.

[214] M. Weiser, A. Demers, and C. Hauser. "The Portable Common Runtime Approach to Interoperability". In: *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*. SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989, pages 114–122. ISBN: 0-89791-338-8. DOI: 10.1145/74850.74862.

[215] 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. Genova, Italy: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328443.

[216] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. "Initialize Once, Start Fast: Application Initialization at Build Time". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360610.

[217] Mario Wolczko. *self includes: Smalltalk*. Presented at the Workshop on Prototype-Based Languages, ECOOP'96, Linz, Austria. 1996.

[218] Mario Wolczko, Ole Agesen, and David Ungar. "Towards a Universal Implementation Substrate for Object-Oriented Languages". In: *OOPSLA workshop on Simplicity, Performance, and Portability in Virtual Machine Design*. 1999.

[219] Carl Worth and contributors. *Cairo*. 2021. URL: https://www.cairographics.org (visited on 2021-07-11).

[220] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. "An Object Storage Model for the Truffle Language Implementation Framework". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: Association for Computing Machinery, 2014, pages 133–144. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647517.

[221] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. "Practical Partial Evaluation for High-Performance Dynamic Language Runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pages 662–676. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062381.

[222] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

[223] Di Yang, Aftab Hussain, and Cristina Videira Lopes. "From Query to Usable Code: An Analysis of Stack Overflow Code Snippets". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pages 391–402. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901767.

[224] Danny Yoo and Shriram Krishnamurthi. "Whalesong: Running Racket in the Browser". In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 97–108. ISBN: 978-1-4503-2433-5. DOI: 10.1145/2508168.2508172.