Cross-Language Interoperability in a Multi-Language Runtime

MATTHIAS GRIMMER and ROLAND SCHATZ, Oracle Labs, Austria
CHRIS SEATON, Oracle Labs, United Kingdom
THOMAS WÜRTHINGER, Oracle Labs, Switzerland
MIKEL LUJÁN, University of Manchester, United Kingdom

In large-scale software applications, programmers often combine different programming languages because this allows them to use the most suitable language for a given problem, to gradually migrate existing projects from one language to another, or to reuse existing source code. However, different programming languages have fundamentally different implementations, which are hard to combine. The composition of language implementations often results in complex interfaces between languages, insufficient flexibility, or poor performance.

We propose TruffleVM, a virtual machine (VM) that can execute different programming languages and is able to compose them in a seamless way. TruffleVM supports dynamically-typed languages (e.g., JavaScript and Ruby) as well as statically typed low-level languages (e.g., C). It consists of individual language implementations, which translate source code to an intermediate representation that is executed by a shared VM. TruffleVM composes these different language implementations via *generic access*. *Generic access* is a language-agnostic mechanism that language implementations use to access foreign data or call foreign functions. It features language-agnostic messages that the TruffleVM resolves to efficient foreign-language-specific operations at runtime. *Generic access* supports multiple languages, enables an efficient multi-language development, and ensures high performance.

We evaluate *generic access* with two case studies. The first one explains the transparent composition of JavaScript, Ruby, and C. The second one shows an implementation of the C extensions application programming interface (API) for Ruby. We show that *generic access* guarantees good runtime performance. It avoids conversion or marshalling of foreign objects at the language boundary and allows the dynamic compiler to perform its optimizations across language boundaries.

CCS Concepts: • **Software and its engineering → Dynamic compilers**; **Runtime environments**; Interpreters;

Additional Key Words and Phrases: Cross-language, language interoperability, virtual machine, optimization, language implementation

ACM Reference format:
Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Trans. Program. Lang. Syst.* 40, 2, Article 8 (May 2018), 43 pages.
https://doi.org/10.1145/3201898

Authors' addresses: M. Grimmer and R. Schatz, Oracle Labs Linz, Altenbergerstraße 69, 4040 Linz, Austria; emails: contact@matthiasgrimmer.com, roland.schatz@oracle.com; C. Seaton, Chris Seaton, 6 Boundary Lane, Heswall, Cheshire, CH60 5RR, UK; email: chris.seaton@oracle.com; T. Würthinger, Oracle Labs Switzerland, Bahnhofstraße 100, 8001 Zurich, Switzerland; email: thomas.wuerthinger@oracle.com; M. Luján, School of Computer Science, University of Manchester, Manchester M13 9PL, UK; email: mikel.lujan@manchester.ac.uk.

https://doi.org/10.1145/3201898

1 INTRODUCTION

In large-scale software development, it is common that programmers write applications in multiple languages rather than in a single language [11]. Combining multiple languages allows them to use the most suitable language for a given problem, to gradually migrate existing projects from one language to another, or to reuse existing source code. There is no programming language that is best for all kinds of problems [3, 11]. High-level languages allow representing algorithms in an elegant way but sacrifice low-level features such as pointer arithmetic and raw memory access. Therefore, business logic is typically written in a high-level language such as JavaScript whereas the computational kernels and networking stacks, for example, are written in a low-level language such as C. Programmers use cross-language interfaces to pick the most suitable language for a given part of a problem. Programmers often also combine different languages when migrating software from one language to another. For example, they can gradually port legacy C code to Ruby, rather than having to rewrite the whole project at once. Finally, cross-language interoperability enables programmers to reuse existing libraries in foreign languages. Due to the large body of existing code, it is often not feasible to rewrite libraries in different languages. A more realistic approach is to interface to the existing code, which allows reusing it.

Our scenario of large-scale software development where programmers mix different languages poses challenges for language implementers as well as for application developers:

- *Multi-language development:* Application programmers need an interface that allows them to switch execution from one language to another and to access foreign data and functions. This application programming interface (API) needs to bridge the different languages on a source code level but also needs to bridge different language paradigms and features. Examples are object-oriented and non-object-oriented programming, dynamic and static typing, explicit and automatic memory management, or safe and unsafe memory access.
- *Language implementation composition:* Different implementations of programming languages execute programs differently (e.g., interpretation on a virtual machine (VM) and native execution) and also use, for example, different object model implementations (dynamic objects of JavaScript and byte sequences of C), use different memory models (automatic memory management of the Java Virtual Machine (JVM) and manual memory management in C), or use different safety mechanisms (runtime errors in Java and segmentation faults in C). Cross-language interoperability needs to bridge code that is running on different implementations. For example, a VM needs a mechanism for calling native code and vice versa. Also, this mechanism needs to provide an interface for accessing foreign data and hereby bridge different strategies for memory management and different implementations of object models.

In the following, we analyze three approaches for cross-language interoperability: (1) foreign function interfaces (FFIs), (2) inter-process communication, and (3) multi-language runtimes. We identified these approaches as the most relevant techniques and point out their major limitations, which are *restricted flexibility*, *complex APIs*, and *performance limitations*.

Foreign Function Interfaces. When programming languages are compiled to different object and code models, composition becomes complicated. One needs a mechanism that can integrate foreign code into a host application and therefore bridge the different implementations. Many modern VMs have an FFI that can integrate native code. An FFI links the native binaries into a VM and also offers an API, which allows the programmer to exchange data between the runtime and the native parts of an application.

The result is a mechanism that caters primarily to composing two *specific* languages rather than *arbitrary* languages. Also, the implementation of these interfaces requires runtime support, which

```
1 typedef void* VALUE;
2 void rb_ary_store(VALUE ary, long idx, VALUE val);
```

Listing 1. Function to store a value in to an array as part of the Ruby API.

means that the APIs often depend on the original implementation of the languages. For example, interpreted languages such as Perl, Python, and Ruby provide support for running extension modules written in the lower-level language C, known as *C extensions* or *native extensions*. C extensions are written in C or C++, and are dynamically loaded and linked into the VM of the high-level language as a program runs. The APIs against which these extensions are written often simply provide direct access to the internal data structures of the standard language implementation. For example, Ruby C extensions[1] are written against the API of the original Ruby implementation Matz' Ruby Interpreter (MRI).[2] This API contains functions that allow C code to manipulate Ruby objects at a high level, but also includes routines that let you directly access pointers to internal data such as the character array in a string object. Figure 1 shows the `rb_ary_store` function, which is part of this interface and allows a C extension to store an element into a Ruby array.

Such APIs are simple, powerful, and reasonably efficient, but they only work well for the language implementations for which they were designed. As dynamic languages become more and more popular, they are going to be reimplemented using modern VM technologies such as dynamic or just-in-time (JIT) compilation and advanced garbage collection. These implementations typically use significantly different internal data structures, which makes an implementation of the original APIs for C extensions hard. As a solution, a bridging layer is often introduced between the internal data structures and the C extensions API. However, this layer imposes costs for transforming the optimized data structures to low-level C data and vice versa. As performance is usually the primary goal of using C extensions, such a bridging layer is suboptimal. Therefore, modern implementations of dynamic languages often have limited support for C extensions. For example, the JRuby[3] implementation of Ruby on top of the JVM had limited support for C extensions. It used a bridging layer to transform Java objects to C data and vice versa. This layer was complicated to maintain and the performance was poor. Eventually, the JRuby team removed the C extensions support completely.[4,5]

Finally, linking native code into the VM impedes compiler optimizations across language boundaries and thus limits performance [52]. Compilers cannot inline or optimize foreign code and need to make conservative assumptions about it. Thus, language boundaries can become a performance bottleneck if multi-language components are tightly coupled.

Inter-Process Communication. Rather than composing language implementations at their implementation level, a message-based inter-process communication treats them as black boxes. Examples are the Common Object Request Broker Architecture [44, 55] or Apache's Thrift [43, 48], which were originally designed for remote procedure call systems but are often used for writing multi-language applications on the same machine. In this scenario, application programmers use a language-agnostic interface description language (IDL) to access objects or to perform operations across languages. This interface marshals data to and from a common wire representation. Programmers who use this approach need to learn and to apply an IDL, which adds a usability

[1] *Ruby Language*, Yukihiro Matsumoto, 2015: https://www.ruby-lang.org.
[2] MRI stands for Matz' Ruby Interpreter, after the creator of Ruby, Yukihiro Matsumoto.
[3] *JRuby*, Charles Nutter and Thomas Enebo and Ola Bini and Nick Sieger and others, 2015: http://jruby.org.
[4] *Ruby Summer of Code Wrap-Up*, Tim Felgentreff, 2015: http://blog.bithug.org/2010/11/rsoc.
[5] *JRuby C Extensions: CRuby extension support for JRuby*, GitHub repository, 2015: https://github.com/jruby/jruby-cext.

```
1 struct S {
2   int value;
3 };
4 struct S * obj = //...
```

Listing 2. C source code.

```
1 // JavaScript can seamlessly access a C struct
2 var a = obj.value;
```

Listing 3. JavaScript source code.

burden. Also, inter-process communication imposes a performance overhead. The marshalling of data introduces a copying overhead whenever language implementations exchange data.

Lightweight remote procedure calls [7] optimize this approach for communication between protected domains on the same machine, which reduces the heavy-weight marshalling of data. However, language implementations still need to treat each other as black boxes and the compiler cannot optimize a program across language boundaries.

Multi-Language Runtimes. Another approach for cross-language interoperability is to run all language implementations on a shared VM. For example, Microsoft's Common Language Runtime (CLR) [10, 15, 42] as well as RPython [9] are runtimes that can host implementations for different languages.

The CLR composes the individual languages by compiling them to a shared intermediate representation (IR). It uses a shared set of IR operations and a shared representation of data for all language implementations, which enables interoperability. A language implementation on top of the CLR is bound to use the Common Type System (CTS) of the CLR. We are convinced that a multi-language runtime could be more flexible. First, the CLR does not support integrating low-level, statically compiled languages like C directly. Native code is integrated via an FFI-like interface. Second, the internal data representation is generic and cannot be optimized for an individual language. Since efficient data representations and data accesses are critical for the performance of an application, we consider this a limiting factor.

RPython uses a different approach in which languages are composed on a very fine granularity (e.g., Python and Prolog [3] or Python and the Hypertext Preprocessor (PHP) [5]). However, we consider this approach as too inflexible because it can only compose a certain pair of languages.

We are convinced that a multi-language runtime is a promising approach for executing multi-language applications efficiently. In this article, we propose TruffleVM, which can execute and combine multiple programming languages. Our runtime features a language-agnostic and flexible mechanism for cross-language interoperability, i.e., it can compose arbitrary languages rather than only a fixed set of languages. We evaluate TruffleVM with three different languages: JavaScript, Ruby, and C. It can execute high-level managed languages as well as low-level unmanaged languages on the same VM and combines the languages with each other.

We approach cross-language interoperability in three steps:

(1) *Composition on the language level:* We combine JavaScript, Ruby, and C by using two different approaches. First, we show a seamless approach for multi-language development. Multi-language applications can access foreign objects and can call foreign functions by simply using the operators of the host language. For example, a field of a C `struct` (Listing 2) can be accessed from a JavaScript program (Listing 3) by just using the JavaScript operator for field accesses. Note that no specific API and no boiler-plate code has to be used.

Second, we combine different languages using an existing FFI. We implement the C extensions API for Ruby on top of TruffleVM. Our solution is source-compatible with the existing Ruby API. Thus, our system is able to run existing, almost unmodified legacy C extensions for Ruby.

(2) *Composition of language implementations:* We present a language-agnostic mechanism for operations on foreign data or code, which we call *generic access*. *Generic access* enables language implementations to efficiently exchange data or code across languages and to bridge possibly different type systems and their semantics.

(3) *Optimization across language boundaries:* Language boundaries often add a performance overhead because the cross-language interface has to marshal data, because language implementations need to use language-agnostic (and therefore less optimized) data representations, and because compilers cannot optimize an application across language boundaries.

TruffleVM executes different languages using abstract syntax tree (AST) interpreters. *Generic access* combines the ASTs of multi-language code, which allows direct access to foreign objects, inlining across language boundaries, and therefore removing the compilation barrier at the language boundaries. Also, this technique allows each language implementation to use data structures that meet the requirements of the individual language best. For example, the C implementation can allocate raw memory on the native heap while the JavaScript implementation can use dynamic objects on a managed heap.

The main contribution of this article (which is an extension of Refs. [25] and [26]) is a novel approach for the seamless composition of programming languages by means of *generic access* to foreign data and code. We demonstrate this idea by describing TruffleVM. The scientific contributions of this article can be grouped into two categories:

— *Language implementation composition:* We present a novel approach for accessing foreign data and functions in a language-agnostic way, which we call *generic access* [25, 26]. *Generic access* is independent of languages, data representations, and calling conventions. It can compose arbitrary languages rather than a fixed set of languages. Multi-language code that runs on top of TruffleVM is interoperable. Also, *generic access* guarantees high performance of multi-language applications. It can directly access any data representation and does not have to marshal data at language boundaries. Moreover, it enables the compiler to optimize and inline across language boundaries.

— *Extensive evaluation by case studies:* We evaluate *generic access* with a case study in which we compose JavaScript, Ruby, and C [25] in single-threaded applications. We list the different language paradigms and semantics and explain how we bridge these differences. We evaluate the performance of multi-language applications using non-trivial multi-language benchmarks.

We furthermore evaluate *generic access* with a second case study, which is an implementation of the C extensions API for Ruby [26]. When running real-world C extensions, our evaluation shows that they run faster than natively compiled C extensions that interface to conventional implementations of Ruby.

2 SYSTEM OVERVIEW

We build on our previous work on Truffle and Graal and use the *Truffle Language Implementations (TLIs)* for JavaScript, Ruby, and C for our case studies. Truffle [61] is a platform for implementing high-performance language implementations as AST interpreters in Java. These AST interpreters model constructs of the guest language as nodes. These nodes build a tree (i.e., an AST) that

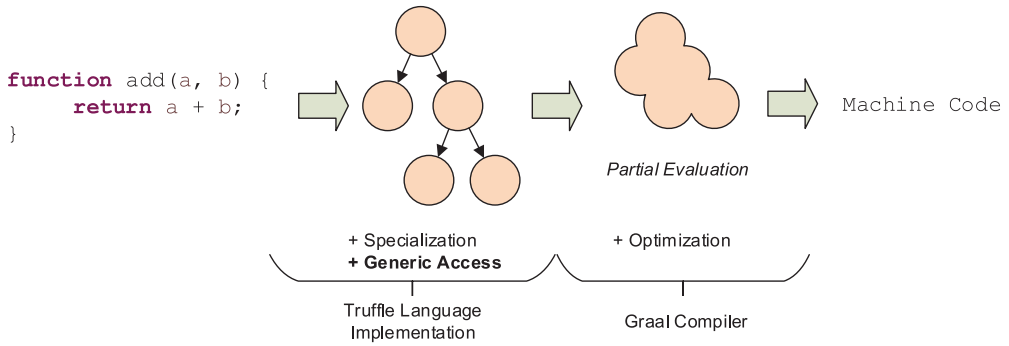

Fig. 1. Parsing guest languages to Truffle ASTs and dynamic compilation using Graal.

represents the program to be interpreted. Each node extends a common base class `Node` and has an `execute` method, which implements the semantics of the corresponding language construct. By calling these methods, the whole AST is evaluated. The language implementer designs the AST nodes and provides facilities to build the AST from the guest language code. Language developers write language-specific parts (i.e., AST nodes for all operations of a language) when implementing a language (see left part of Figure 1).

Truffle AST nodes can speculatively rewrite themselves with *specialized* variants [62] at runtime, e.g., based on profile information obtained during execution such as type information. Nodes specialize on a subset of the semantics of an operation, i.e., they replace themselves with a simpler (and often faster) implementation. Language developers can implement this self-optimization via tree rewriting for their nodes and use it as a general mechanism for dynamically optimizing code at runtime. If these speculative assumptions turn out to be wrong, the specialized tree can be transformed to a more generic version that provides functionality for all possible cases.

Concrete examples of specializations are:

Type Specialization. Operators in dynamic languages often have complex semantics. The behavior of an operation can, for example, depend on the types of the operands. Hence, such an operator needs to check the types of its operands and choose the appropriate version of the operator. However, for each instance of a particular operator in a guest language program, it is likely that the types of its operands do not change at runtime. Truffle's self-optimization capability allows us to replace the full implementation of an operation with a specialized version that speculates on the types of the operands being constant. This specialized version then only includes the code for this single case. For example, consider an add operation of a dynamic language (e.g., JavaScript) that has different semantics depending on the types of its operands. Truffle trees can adapt themselves according to type feedback, e.g., a general add operation can replace itself with a faster integer-add operation if its operands have been observed to be integers.

Type specialization is also used to efficiently implement object accesses. For example, consider a property access in JS (see Figure 2). TruffleJS specializes the property read on the shape of object `obj` [58]. This specialization assumes that the shape of the object is constant and directly accesses the object member `value`.

If the optimistic specialization of an operator fails at runtime, the specialized node changes back to a more generic version.

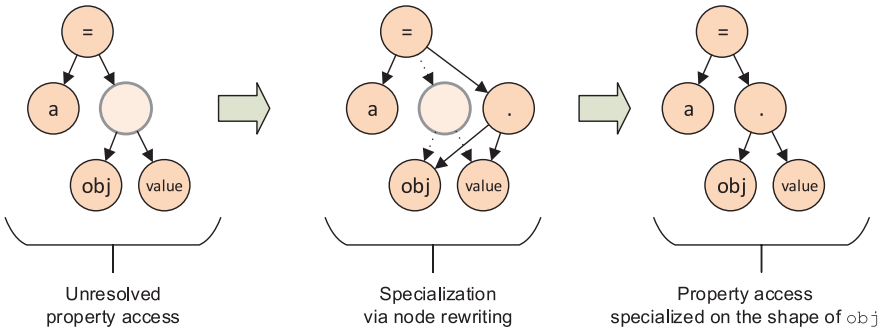

Fig. 2. Specialization of a property access `a = obj.value` in TruffleJS.

Polymorphic Inline Caches. Truffle's self-optimization capability allows building up polymorphic inline caches [30] at runtime. An inline cache is an optimization that improves the performance of runtime method binding by caching the target method of a previous lookup at the call site. In Truffle, an inline cache is built by chaining nodes. Each node in this chain then checks whether the cached target matches and eventually executes the specialized subtree for this target. If the target does not match, the chain delegates the handling of the operation to the next node. When the chain reaches a predefined length, the whole chain replaces itself with a single node that can handle the fully megamorphic case.

Resolving Operations. Operations of a TLI might include a resolving step that happens at runtime. For example, a TLI can resolve and cache the target of a function call lazily at runtime. This lazy resolving step is implemented with self-optimization, which in this case replaces the node of an unresolved call operation at runtime by its resolved version. In subsequent executions, this node need not be resolved again.

After an AST has become stable (i.e., when no more rewritings occur) and when the execution frequency has exceeded a predefined threshold, Truffle partially evaluates [60] the tree and dynamically compiles it to optimized machine code (see right part of Figure 1). Partial evaluation and all compiler optimizations are transparent to the Truffle language implementer and happen automatically.

Truffle uses the Graal compiler [13, 14, 23, 49–51] for dynamic compilation. Partial evaluation means that the Graal compiler inlines all node execution methods of a tree into a single method, assuming that the tree remains stable. This allows the compiler to remove all the virtual dispatches between the `execute` methods of the AST nodes and to inline them. Inlining produces a combined compilation unit for the whole tree. The Graal compiler can then apply its aggressive optimizations over the whole tree, which results in highly efficient machine code. This special form of interpreter compilation is an application of *partial evaluation* to generate compiled machine code from a specialized interpreter [18].

The compiler inserts deoptimization points [31] in the machine code where the speculative assumptions about the tree are checked. Every control flow path that finds such an assumption violated transfers back from the compiled machine code to the interpreted AST, where specialized nodes can be reverted to a more generic version. This is called deoptimization.

The TruffleVM is a modification of the HotSpot VM: it adds the Graal compiler, Truffle, and the TLIs, but reuses all other parts of the HotSpot VM, including the garbage collector and the interpreter. Figure 3 shows the layers of TruffleVM.

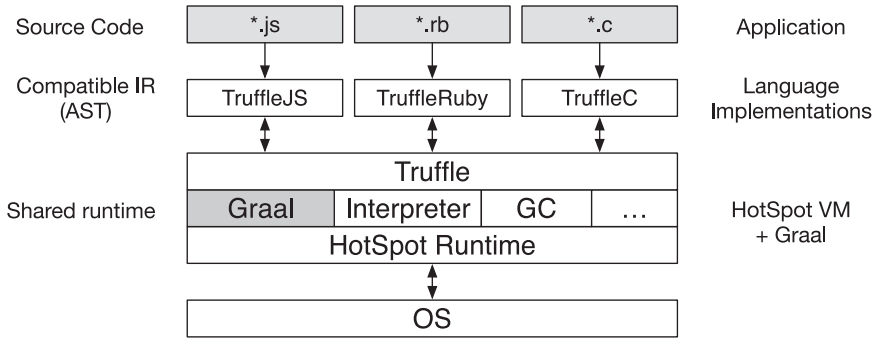

Fig. 3. Layers of the TruffleVM: TruffleJS, TruffleRuby, and TruffleC are hosted by the Truffle framework on top of the HotSpot VM (using Graal as a JIT compiler).

For our work, we modified TLIs and extended the Truffle framework itself (left part of Figure 1); no changes were necessary to the Graal compiler or any other parts of the HotSpot VM (right part of Figure 1). For our case studies, we used TLIs for JavaScript (TruffleJS), Ruby (TruffleRuby), and C (TruffleC).

2.1 TruffleJS

TruffleJS is an implementation of JavaScript on top of Truffle. It was originally Truffle's proof of concept. The closed-source implementation is available as a binary on Oracle's Technology Network.[6]

JavaScript is a dynamically typed and prototype-based scripting language. It allows an object-oriented, an imperative, as well as a functional programming style, which makes it a good candidate for the evaluation and the case study in this article. TruffleJS uses AST specialization, e.g., for specializing operations according to the observed dynamic types of their operands.

TruffleJS is a state-of-the-art JavaScript engine that implements the ECMAScript 2016 standard. It passes 93% of the ECMAScript 2016 standard test suite [53]. At the point of writing this article, the failing features were: TruffleJS did not implement the `unicode` flag of RegExp; failures in corner case tests in Shared Array Buffers, Async/Await, and Destructing Assignments. Its performance was evaluated in Ref. [58] using a selected set of benchmarks from the Octane benchmark suite. Figure 4 summarizes the performance evaluation of Ref. [58] for JavaScript. The y-axis shows the performance of Google's V8,[7] Mozilla's Spidermonkey,[8] and Nashorn as included in JDK 8u5[9] relative to TruffleJS where the outermost lines show the minimum and maximum performance and the inner dot shows the average performance. The x-axis shows the different language implementations.

Google's V8 is between 210% faster and 67% slower (52% faster on average) than TruffleJS; Mozilla's Spidermonkey is between 160% faster and 22% slower (54% faster on average) than TruffleJS; Nashorn is between 12% faster and 96% slower (74% slower on average) than TruffleJS. A more detailed description of TruffleJS can be found in Ref. [58].

[6] *JavaScript implementation as part of GraalVM*, Oracle, 2017: http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html.
[7] *V8 JavaScript Engine*, Google, 2015: http://code.google.com/p/v8.
[8] *SpiderMonkey JavaScript Engine*, Mozilla Foundation, 2015: http://developer.mozilla.org/en/SpiderMonkey.
[9] *Nashorn JavaScript Engine*, Oracle, 2015: http://openjdk.java.net/projects/nashorn.

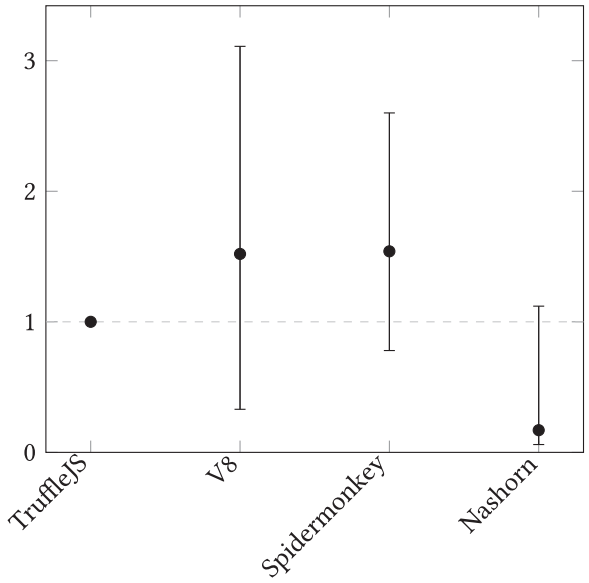

Fig. 4. Relative speedup of different JavaScript implementations compared to TruffleJS; results taken from Ref. [58] (higher is better).

2.2 TruffleRuby

TruffleRuby is an implementation of Ruby on top of Truffle. Ruby is a dynamically typed object-oriented language that is inspired by Smalltalk and Perl. TruffleRuby reuses the parser from JRuby. Otherwise, however, the two systems have little in common. TruffleRuby (also called JRuby+Truffle in Refs. [26] and [47]) should be considered entirely separate from JRuby for this discussion. TruffleRuby is an open source project and available on GitHub.[10]

TruffleRuby is a state-of-the-art Ruby engine that aims to be highly compatible with the standard implementation of Ruby, MRI, version 2.3.3. TruffleRuby passes 98% (language) and 94% (core library) of the *Ruby spec* standard test suite. At the point of writing this article, the failing features were: TruffleRuby does not implement continuations and `callcc`, `fork` of the TruffleRuby interpreter, and refinements. Figure 5 summarizes the performance evaluation of Ref. [58] for TruffleRuby. The y-axis shows the performance of MRI, Rubinius[11,12], JRuby, and Topaz[13] relative to TruffleRuby where the outermost lines show the minimum and maximum performance and the inner dot shows the average performance. The x-axis shows the different language implementations. This evaluation uses the Richards and DeltaBlue benchmarks from the Octane suite, a neural-net, and an n-body simulation. MRI is between 63% and 97% slower (92% slower on average) than TruffleRuby; Rubinius is between 24% and 97% slower (83% slower on average) than TruffleRuby; JRuby is between 61% and 94% slower (84% slower on average) than TruffleRuby; Topaz is between 1% and 87% slower (71% slower on average) than TruffleRuby. A more detailed description of TruffleRuby can be found in Refs. [26] and [47].

[10] *Ruby implementation on top of Truffle*, Oracle, 2017: https://github.com/graalvm/truffleruby.
[11] *An implementation of Ruby*, Rubinius, 2015: http://rubini.us.
[12] *Rubinius*, GitHub repository, 2015: https://github.com/rubinius/rubinius.
[13] *Topaz Project*, GitHub repository, 2015: https://github.com/topazproject/topaz.

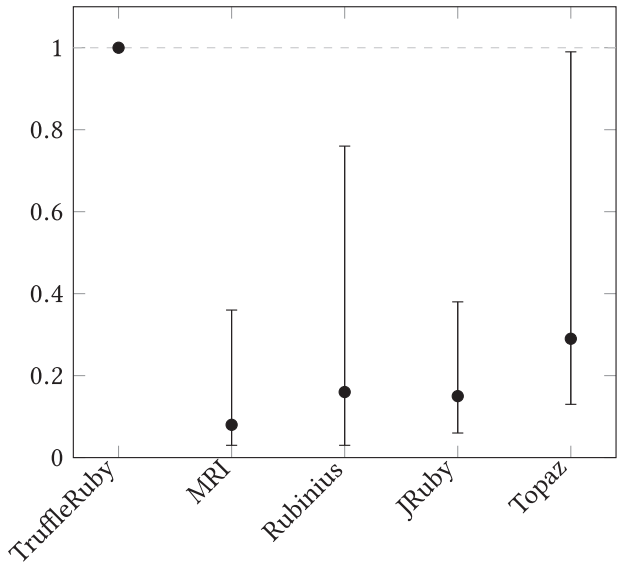

Fig. 5. Relative speedup of different Ruby implementations compared to TruffleRuby; results taken from Ref. [58] (higher is better).

2.3 TruffleC

TruffleC [22] is an implementation of C on top of Truffle. It executes C code and makes heavy use of Truffle's self-optimization capability: TruffleC uses polymorphic inline caches [30] to efficiently handle function pointer calls and profiles branch probabilities to optimistically remove dead code. It also profiles runtime values and replaces them with constants if they do not change over time.

A C program running on top of TruffleC can switch execution from TruffleC to a native function (e.g., one that is part of the standard C library) using Graal's native function interface (GNFI) [23]. When switching from the TruffleC interpreter (that is written in Java) to native code, GNFI allows passing parameters from Java to a native function. Besides passing parameters, TruffleC can also exchange data of the running C program via pointers. TruffleC allocates C data on the native heap rather than on the garbage-collected Java heap and uses the same data alignment as conventional C compilers do, which allows sharing allocations (e.g., structs, unions, and arrays) between TruffleC and native code. Also, this allows TruffleC to support pointer arithmetic and to replicate the same behavior as conventionally compiled C code. To access the native heap, TruffleC uses the Java Unsafe API (available under restricted access in the OpenJDK). The Unsafe API provides functionality to manually allocate memory on the native heap and to load data from it and store data into it.

TruffleC uses `CAddress` Java objects to implement pointers. `CAddress` objects that point to an allocation on the native heap wrap a raw memory address as a 64-bit value [27]. `CAddress` objects that point to C functions (i.e., a Truffle AST) use a Java object reference. Also, a `CAddress` attaches type information to pointers, which makes *generic access* for TruffleC (see Section 4.1) possible.

TruffleC aims to support the C99 standard [32], however, it is not yet fully complete. It does not yet have support for flexible array members, variable-length automatic arrays, designated initializers, and compound literals. *Flexible array members*[14] allow defining a C `struct` that has an array

[14] *Arrays of length zero*, GCC the GNU Compiler Collection, 2015: https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html.

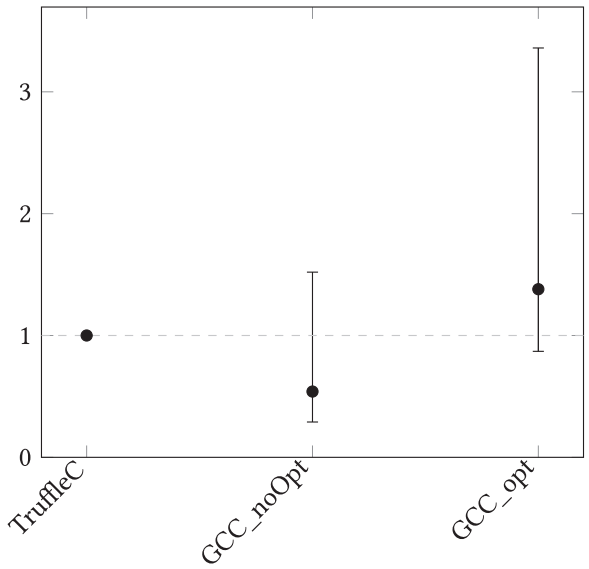

Fig. 6. Relative speedup of code generated by GCC compared to TruffleC; results taken from Ref. [24] (higher is better).

member without a given dimension. *Variable-length automatic arrays*[15] are array data structures whose length is determined at runtime. *Designated initializers*[16] allow assigning values to arrays or `structs` in any order. *Compound literals*[17] look like casts that contain an initialization. The initialization value is specified as a list of all values to be assigned. Adding these features would only require additional engineering effort, i.e., TruffleC has no conceptual restrictions in this respect. Adding support for these missing features is planned as future work.

TruffleC's performance was evaluated in Ref. [24] using a selected set of benchmarks from the computer language benchmarks game.[18] Code generated by GCC[19] without optimizations is between 52% faster and 71% slower (46% slower on average) than TruffleC; code generated with the highest optimization level of GCC is between 236% faster and 13% slower (38% faster on average) than TruffleC.

Figure 6 summarizes the performance evaluation of Ref. [24] for TruffleC. The y-axis shows the performance of code generated by GCC[20] without optimizations (GCC_noOpt) and code generated by GCC with the highest optimization level (GCC_opt) relative to TruffleC where the outermost lines show the minimum and maximum performance and the inner dot shows the average performance. The x-axis shows the different language implementations. This evaluation

[15] *Arrays of variable length*, GCC the GNU Compiler Collection, 2015: https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html.
[16] *Designated initializers*, GCC the GNU Compiler Collection, 2015: https://gcc.gnu.org/onlinedocs/gcc/Designated-Inits.html.
[17] *Compound literals*, GCC the GNU Compiler Collection, 2015: https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Compound-Literals.html.
[18] *Computer Langauge Benchmarks Game*, 2016: http://benchmarksgame.alioth.debian.org.
[19] *GNU C Compiler*, Free Sofware Foundation Inc., 2016: https://gcc.gnu.org.
[20] *GNU C Compiler*, Free Sofware Foundation Inc., 2016: https://gcc.gnu.org.

uses a selected set of benchmarks from the computer language benchmarks game.[21] GCC_noOpt is between 52% faster and 71% slower (46% slower on average) than TruffleC; GCC_opt is between 236% faster and 13% slower (38% faster on average) than TruffleC. A more detailed description of TruffleC can be found in [22, 24, 26, 27].

3 GENERIC ACCESS

TLIs can use different layouts for objects. For example, the JavaScript implementation allocates data on the Java heap, whereas the C implementation allocates data on the native heap. Given the different layouts, each TLI uses language-specific AST nodes to access them. In the context of this article, we use the term *object* for a non-primitive entity of a user program, which we want to share across different TLIs. Examples include data references (such as JavaScript object references, Ruby object references, or pointers to C allocations) as well as function references. If a host language L_{Host} accesses one of its own objects, we call this a regular object access. For example, if the Ruby implementation accesses a Ruby object, the object is considered a regular object. If it accesses an object of a foreign language $L_{Foreign}$, we call this a foreign object access. For example, if Ruby (L_{Host}) accesses a C structure, the C structure is considered a foreign object (C is $L_{Foreign}$). *Object accesses* are operations that an L_{Host} can perform on objects, e.g., method calls, property accesses, or field accesses.

Foreign Object Access. TLIs specialize object accesses on the receiver type. For example, TruffleJS specializes a property access depending on the shape of the receiver object [58] (cf. Section 2). We extend these specializations by a case for *foreign objects*. This specialization triggers when a L_{Host} encounters a foreign object at runtime and a regular object access operation cannot be used. In such cases, L_{Host} uses *generic access* to access the foreign object:

Generic access can access every object that implements the common `TruffleObject` interface. We implement this interface as a set of *messages*. To access a foreign object via *generic access*, L_{Host} specializes the object access to a language-agnostic node that sends a message. For example, an L_{Host} can access the members of a `TruffleObject` by *Read* and *Write* messages. The left part of Figure 7 shows a JavaScript AST snippet that was specialized to *generic access* and reads the `value` property of a foreign object `obj` using a *Read* message.

Message Resolution. The first execution of a message node (e.g., the gray node in Figure 7) does not directly access the receiver object but triggers *message resolution*: TruffleVM resolves the message to a foreign-language-specific AST snippet, i.e., it lets $L_{Foreign}$ produce a foreign-language-specific AST snippet (green nodes in Figure 7) that is inserted into the host AST as a replacement for the message. This AST snippet depends on the type of the receiver and contains foreign-language-specific nodes for executing the message on the receiver.

In order to notice an access to an object of a previously unseen foreign language, message resolution inserts a guard into the AST that checks the receiver's language before it is accessed. As can be seen in Figure 7, message resolution inserts a *C struct access* node (a TruffleC-specific AST node that accesses a struct member; in Figure 7, we use the abbreviated label "`->`" for this node). Before the AST accesses `obj`, it checks if `obj` is really a C object (*is C?*). If `obj` is suddenly an object of a different language, the execution falls back to sending a *Read* message again, which will then be resolved to a new AST snippet for this language. An object access is *language polymorphic* if it has varying receivers originating from different languages. In the language-polymorphic case, TruffleVM links the different language-specific AST snippets into a chain similar to an inline

[21] *Computer Langauge Benchmarks Game*, 2016: http://benchmarksgame.alioth.debian.org.

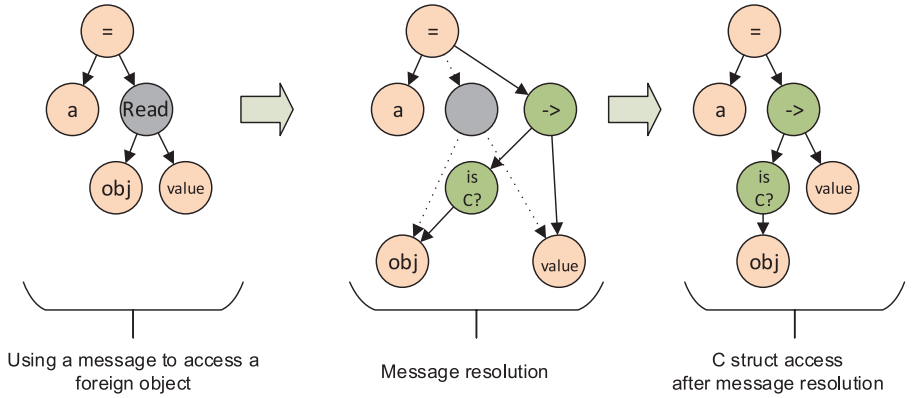

Fig. 7. *Generic access* uses messages to access a foreign object; Message resolution replaces the *Read* message by a direct access.

cache [30] and therefore avoids a loss in performance. In the megamorphic case, TruffleVM checks the language of the foreign object and dispatches a call to the foreign-language-specific AST snippets that can access the object. In our case study with three languages, no object access is *language polymorphic*.

Message resolution only affects the application's performance upon the first execution of an object access. By generating AST snippets for accessing foreign objects, we avoid compilation barriers between languages. This allows the compiler to optimize object accesses or to inline method calls, even if the receiver is a foreign object. Widening the compilation unit across different languages is important [2, 49] as it enables the compiler to apply optimizations to a wider range of code.

Primitive Types. The TruffleVM defines a set of *shared primitive types* to exchange primitive values across languages. We refer to such values as *shared primitives*. The shared primitive types include all Java built-in types, such as all number classes that extend `java.lang.Number` and also the `java.lang.String` type. A TLI maps its language-specific primitive values to *shared primitive* values and exchanges them as language-neutral values. Vice versa, a TLI maps *shared primitive* values to language-specific values. Using this set of types works well for TLIs because TLIs are themselves written in Java; hence, the TLIs already map the primitive types of the guest language to Java types.

We identified the following properties of TruffleVM as critical requirements for *generic access*:

— *Compatible and adaptive ASTs:* Message resolution embeds AST snippets from a foreign TLI into the AST of a host TLI at runtime, which requires that the ASTs are compatible, inter-mixable, and can rewrite themselves.
— *Sharable data:* The data structures used by TLIs to represent the data of an application need to be accessible by all language implementations.
— *Dynamic compilation:* Graal compiles the ASTs of a (possibly multi-language) program to highly efficient machine code at runtime. It inserts deoptimization points that invalidate the machine code whenever *generic access* would change the AST of an application.

4 CASE STUDIES

In this section, we present two case studies. First, we show an implementation of *generic access* for JavaScript, Ruby, and C and explain how we compose these languages. Second, we describe how

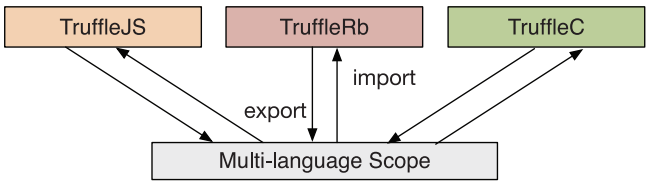

Fig. 8. TLIs can import and export objects from and to a multi-language scope.

```
1 #include <truffle.h>
2
3 typedef struct {
4   int value;
5 } S;
6
7 S *obj = //...
8 Export("obj", obj);
```

Listing 4. This C snipped exports the variable `obj` which points to a `struct` of type `S`.

```
1 var obj = Truffle.import("obj");
2 var a = obj.value;
```

Listing 5. This JavaScript snipped imports a variable `obj` and accesses its `value` member.

we implement the C extensions API for Ruby. We use *generic access* to provide an implementation for this interface, which allows us to run existing production-code.

4.1 Interoperability between JavaScript, Ruby, and C

TruffleVM can execute programs that are written in multiple languages. Programmers use different files for different programming languages. For example, if parts of a program are written in JavaScript and C, these parts are in different files. Distinct files for each programming language allow us to reuse the existing unmodified parsers of each language implementation to transform source code to Truffle ASTs (TruffleJS uses the parser of Nashorn, TruffleRuby uses JRuby's parser, and TruffleC uses the Clang frontend). The combination of multi-language code parts in a single source file is out of scope for this work, although it would be possible in principle [3, 29].

Programmers can export data and functions to a multi-language scope and also import data and functions from this scope. Figure 8 shows that all TLIs access this multi-language scope, which allows programmers to share data and functions among languages. JavaScript, Ruby, and C use Truffle built-ins to export and import data to and from the multi-language scope. For example the C code of Listing 4 exports the C `struct obj` to the multi-language scope and the JavaScript code of Listing 5 imports it.

Implicit Foreign Object Accesses. TruffleVM allows programmers to access foreign objects transparently. If the host TLI encounters a foreign object at runtime and a regular object access operation cannot be used, then the host TLI specializes the object access to *generic access*. It hereby maps the *host-language-specific operation* to a *language-independent message*, which is then mapped back to a *foreign-language-specific operation* by the foreign TLI. In the following, we describe this transformation.

TLIs transform source code to a tree of nodes, i.e., an AST. N^A and N^B define finite sets of nodes of TLIs A and B. Each node has $r : N^A \rightarrow \mathbb{N}$ children, where $\mathbb{N}$ denotes the set of natural numbers. If $n \in N^A$ is a node, then $r(n)$ is the number of its children. We call nodes with $r = 0$ *leaf* nodes. For example, nodes that represent constants and labels are *leaf* nodes. An AST $t \in T_{N^A}$ is a tree of nodes $n \in N^A$. By $n(t_1, \ldots, t_k)$, we denote a tree with root node $n \in N^A$ and k sub-trees $t_1, \ldots, t_k \in T_{N^A}$, where $k = r(n)$.

For this case study, we define the following set of messages, which are modeled as Truffle nodes N^{Msg}:

$$N^{\text{Msg}} = \{\text{Read}, \text{Write}, \text{Execute}, \text{Unbox}, \text{IsNull}\} \tag{1}$$

If TLI A encounters a foreign object at runtime and a regular object access operation cannot be used, then TLI A maps the AST with the language-specific object access $t \in T_{N^A}$ to an AST with a language-agnostic object access $t' \in T_{N^A \cup N^{\text{Msg}}}$ using the function f_A:

$$T_{N^A} \xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \tag{2}$$

We use the function f_A when specializing an object access to *generic access*. The tree $t' \in T_{N^A \cup N^{\text{Msg}}}$ consists of language-specific nodes N^A and message nodes N^{Msg}. The other parts of the AST t remain unchanged. A host language that accesses foreign objects has to define this function f.

In the following, we describe the messages $n \in N^{\text{Msg}}$, which we use for this case study. The sub-trees $t_1, \ldots, t_k \in T_{N^A \cup N^{\text{Msg}}}$ of $n(t_1, ..., t_k)$ evaluate to the arguments of the message n.

— *Read:* TLIs use the *Read* message to access a field of a foreign object or an element of a foreign array. It can also be used to access methods of classes or objects, i.e., to look up executable methods from classes and objects.

$$\text{Read}(t_{\text{rec}}, t_{\text{id}}) \in T_{N^A \cup N^{\text{Msg}}} \tag{3}$$

The first subtree t_{rec} denotes the receiver of the *Read* message; the second subtree t_{id} denotes the name of the field or the index of the array element.

Consider the example in Figure 7 (a property access `obj.value` in JavaScript, see Listing 5). The function f_{JS} maps the JavaScript-specific object access *JSReadProperty* to a *Read* message at runtime if JavaScript suddenly encounters a foreign object.

$$\text{JSReadProperty}(t_{\text{obj}}, t_{\text{value}}) \xmapsto{f_{JS}} \text{Read}(t_{\text{obj}}, t_{\text{value}}) \tag{4}$$

— *Write:* A TLI uses the *Write* message to set the field of a foreign object or the element of a foreign array. It can also be used to add or change the methods of classes and objects.

$$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{\text{val}}) \in T_{N^A \cup N^{\text{Msg}}} \tag{5}$$

The first subtree t_{rec} denotes the receiver of the *Write* message, the second subtree t_{id} the name of the field or the index of the array element, and the third subtree t_{val} the written value.

— *Execute:* TLIs use an *Execute* message to execute a foreign method or function.

$$\text{Execute}(t_{\text{f}}, t_1, \ldots, t_i) \in T_{N^A \cup N^{\text{Msg}}} \tag{6}$$

The first subtree t_{f} denotes the function/method itself, the other subtrees $t_1, \ldots, t_i$ denote the arguments.

— *Unbox:* Programmers often use an object type to wrap a value of a primitive type in order to make it look like a real object. An *Unbox* message unwraps such an object and produces

a primitive value. TLIs use this message to unbox a boxed value whenever a primitive value is required.

$$\text{Unbox}(t_{\text{rec}}) \in T_{N^A \cup N^{\text{Msg}}} \tag{7}$$

The subtree t_{rec} denotes the receiver object.

— *IsNull:* Many programming languages use `null`/`nil` for an undefined, uninitialized, empty, or meaningless value. The *IsNull* message allows the TLI to do a language-agnostic `null`-check.

$$\text{IsNull}(t_{\text{rec}}) \in T_{N^A \cup N^{\text{Msg}}} \tag{8}$$

The subtree t_{rec} denotes the receiver object.

As part of *message resolution*, TruffleVM maps the host AST with a language-agnostic access $t' \in T_{N^A \cup N^{\text{Msg}}}$ to an AST with a foreign-language-specific access $t'' \in T_{N^A \cup N^B}$ using the function g_B, which is defined by $\text{L}_{\text{Foreign}}$:

$$T_{N^A \cup N^{\text{Msg}}} \xrightarrow{g_B} T_{N^A \cup N^B} \tag{9}$$

Message resolution produces an AST that consists of nodes $N^A \cup N^B$. The other nodes of t' remain unchanged. With respect to the example in Figure 7, TruffleVM uses the function g_C (defined by TruffleC) and replaces the *Read* message with a C-specific `struct` access operation upon its first execution. If the receiver is a pointer to a C `struct`, then TruffleC maps a *Read* message to a *CMemberRead* node (node "->" in the AST):

$$\text{Read}(t_{\text{obj}}, t_{\text{value}}) \overset{g_C}{\longmapsto} \text{CMemberRead}(\text{IsC}(t_{\text{obj}}), t_{\text{value}}) \tag{10}$$

The result is a JavaScript AST (orange nodes in Figure 7) that embeds a C access operation $t'' \in T_{N^{\text{JS}} \cup N^C}$ (green nodes). During further executions of the AST, the receiver object is accessed directly rather than by a message. Since the value of `obj` might change at runtime to represent an object of a language other than Cm, message resolution inserts a guard into the AST that checks the receiver's type before it is accessed. This is shown in Figure 7 where message resolution inserts a node that checks if `obj` is a C object.

We can achieve a seamless foreign object access by mapping *host-language access operations* to *messages*, which are then mapped back to *foreign-language-specific operations*. The function f_A maps the AST t to an AST $t' \in T_{N^A \cup N^{\text{Msg}}}$ that uses a message to access the foreign object. The foreign language defines the function g_B that maps t' to an AST $t'' \in T_{N^A \cup N^B}$ with a foreign-language-specific object access:

$$\begin{aligned} T_{N^A} &\xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \\ T_{N^A \cup N^{\text{Msg}}} &\xrightarrow{g_B} T_{N^A \cup N^B} \end{aligned} \tag{11}$$

TruffleVM composes TLIs automatically by composing f_A and g_B at runtime:

$$g_B \circ f_A : T_{N^A} \to T_{N^A \cup N^B} \tag{12}$$

It creates an AST $t'' \in T_{N^A \cup N^B}$ where the main part is specific to language A and the foreign object access is specific to language B. When composing f_A and g_B three different cases can occur:

(1) If g_B is defined for $t' \in T_{N^A \cup N^{\text{Msg}}}$, a foreign object can be accessed seamlessly. The language B can replace the language-agnostic object access with a foreign-language-specific access.
(2) If g_B is *not* defined for $t' \in T_{N^A \cup N^{\text{Msg}}}$, we report a runtime error with a high-level diagnostic message. The foreign object access is not supported. For example, if JavaScript accesses the length property of a C array, we report an error. C cannot provide length information for arrays.

```
1 var arr = new Array(5);
2 Truffle.export("jsArray", arr);
```

Listing 6. Array allocation in JavaScript.

```
1 #include <truffle.h>
2
3 int * arr = (int*) Import("jsArray");
4 int length = Read_int32(arr, "length");
```

Listing 7. C code accessing the length property of an array.

(3) A foreign object access might not be expressible in A, i.e., one wants to create $t' \in T_{N^A \cup N^{\text{Msg}}}$, but language A does not provide syntax for this access. For example, a C programmer cannot access the length property of a JavaScript array. In this case one has to fall back to an *explicit* foreign object access (see below).

Compared to other approaches that use an explicit API for every interaction with a foreign language, our approach is simpler. It makes the mapping of access operations to messages largely the task of the language implementer rather than the task of the application programmer. Programmers are not forced to write boilerplate code as long as an object access can be mapped from language A to language B ($t \overset{f_A}{\longmapsto} t' \overset{g_B}{\longmapsto} t''$) via *generic access*. Only if not otherwise possible, programmers can use *explicit generic access* to access foreign objects.

Explicit Foreign Object Access. A host language might not provide syntax for a specific foreign object access. Consider the JavaScript array `arr` of Listing 6, which is used in a C program. C does not provide syntax for accessing the length property of an array. To overcome this issue, we provided an interface to the programmer that allows using *explicit generic access*. Using this interface, the programmer can fall back to an *explicit* foreign object access. In other words, this interface allows programmers to handcraft the foreign object access of $t' \in T_{N^A \cup N^{\text{Msg}}}$.

Every TLI has an API to use *explicit generic access*. For example, to access the `length` property of a JavaScript array (see Listing 6) from C (see Listing 7), the programmer uses the TruffleC-built-in C function `Read_int32`. The C implementation substitutes this `Read_int32` invocation by a *Read* message.

4.1.1 Implementation of Generic Access. In the following, we discuss how we implement *generic access* for JavaScript, Ruby, and C. We explain how each language maps host-type-specific object accesses to messages and vice versa.

TruffleJS. JavaScript is a prototype-based scripting language with dynamic typing. It is almost completely object-oriented. JavaScript objects are associative arrays that have a prototype, which corresponds to their dynamic type. Object property names are string keys and it is possible to add, change, or delete properties of an object at runtime. In TruffleJS, all objects are shareable in the sense that they support *generic access*. TruffleJS maps property accesses to *Read* and *Write* messages. Functions are first-class objects. TruffleJS maps a function call to an *Execute* message. Method invocations on objects are mapped to a sequence of *Read* and *Execute* message. It maps incoming numeric primitive values to objects of type `Number`, i.e., JavaScript's type for representing number values. Also, TruffleJS unboxes (using the *Unbox* message) boxed foreign primitive values (e.g., Ruby's `Float`) and maps them to objects of type `Number`. `Number` objects support the *Unbox*

```
// Imports a TruffleObject from the multi-language scope
Truffle.import(id);
// Exports an object to the multi-language scope
Truffle.export(id, val);

// Reads from a TruffleObject
Truffle.read(receiver, id);
// Writes to a TruffleObject
Truffle.write(receiver, id, value);

// ...
```

Listing 8. Excerpt of the built-ins (`Truffle`-object methods) for JavaScript.

message, which allows sharing them among other languages. *Unbox* maps the value to a numeric *shared primitive*.

The implementation of *generic access* for TruffleJS also introduces a built-in object called `Truffle` (see Listing 8). The `Truffle` object defines functions for importing and exporting objects from and to the multi-language scope. Also, it defines functions to explicitly access foreign objects. TruffleJS substitutes every call to these functions by a *generic access* or a multi-language scope access.

TruffleRuby. The Ruby language is heavily inspired by Smalltalk; hence, in Ruby there are no primitive types. Every value—including numeric values—is represented as an object. Operations (e.g., arithmetic operations) as well as data access operations (accessing object attributes or array elements) are modeled as function calls on the receiver object. For example, Ruby arrays or hashes provide a setter method `[]=` to set an element of a Ruby array or hash. We map getter and setter invocations (functions `[]` and `[]=`) to *Read* and *Write* messages. In TruffleRuby, all data objects as well as all methods support *generic access* and are therefore sharable. TruffleRuby maps incoming primitive values to objects of numeric type `Fixnum` and `Float` if this is possible without loss of information (e.g., no truncation or rounding). These objects are also sharable with other languages, i.e., they support the *Unbox* message. This message simply maps the boxed value to the corresponding shared primitive. For example, a host language other than Ruby might use an *Unbox* message whenever it needs the object's value for an arithmetic operation.

Similar to TruffleJS, TruffleRuby defines a built-in class `Truffle`, which allows exporting and importing variables to and from the multi-language scope and to explicitly access foreign objects.

TruffleC. TruffleC can share primitive C values, mapped to *shared primitive values*, as well as pointers to C data with other languages. In our implementation, pointers are objects that support *generic access*, which allows them to be shared across all TLIs. TruffleC represents all pointers (i.e., pointers to values, arrays, `structs`, or functions) as `CAddress` objects that wrap a 64-bit value. This value is a pointer to the native heap. Besides the address value, a `CAddress` object also stores type information about the pointee. Depending on the type of the pointee, TruffleC resolves the following messages: A pointer to a C `struct` can resolve *Read*/*Write* messages, which access members of the referenced `struct`. A pointer to an array can resolve *Read*/*Write* messages that access a certain array element. Any pointer can resolve the *IsNull* message, which checks whether the pointer is a `null`-pointer. Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message. TruffleC can bind `CAddress` objects as well as shared foreign objects to pointer variables and uses *generic access* to operate on these foreign objects.

```
int *arr = (int*) Import("arr");
int value = arr[0];
```

Listing 9. C `int` expression that reads from a foreign object.

```
// Imports a TruffleObject from the multi-language scope
void* Import(const char* id);
// Exports a pointer to the multi-language scope
void  Export(const char* id, void* val);

// Reads a 32-bit integer from a TruffleObject
int   Read_int32(void* receiver, const char* id);
// Writes a 32-bit integer to a TruffleObject
void  Write_int32(void* receiver, const char* id, int value);

// ...
```

Listing 10. Excerpt of the `truffle.h` header file.

C is a statically typed language and every expression has a static type. Hence, if TruffleC accesses a foreign object and the access expression has a primitive type, TruffleC tries to convert the result to a value that has the same type as the expression. Consider the following example (Listing 9):

TruffleC converts the result of `arr[0]` to a C `int` value. It directly converts shared primitive values to `int` values if this is possible without loss of information. If the result is a `TruffleObject`, TruffleC uses *Unbox* and then does the conversion. If a conversion is not possible, TruffleC raises a runtime error and reports the type-incompatibility.

The implementation of *generic access* for TruffleC also introduces the header file `truffle.h` (see Listing 10), which defines functions for importing and exporting objects from and to the multi-language scope as well as functions for explicitly accessing foreign objects. There are different versions of these functions for all primitive C types. For example, there is a function `Read_int32` that reads from a foreign object and tries to convert the result to an `int` value. None of the functions that are defined in `truffle.h` have an implementation in C. Instead, TruffleC substitutes every invocation by a *generic access* or a multi-language scope access.

For further reference, we provide a detailed table that lists all mappings from language-specific operations to messages and vice versa.[22] Also, we point interested readers to the Java documentation of the *generic access* API in Truffle.[23]

4.1.2 Different Language Paradigms and Features. In this section, we describe an intuitive approach for bridging the different paradigms and features of JavaScript, Ruby, and C. We focus on these languages and explain how we deal with dynamic and static typing, object-oriented and non-object-oriented programming, explicit and automatic memory management, as well as with safe and unsafe memory accesses. For this discussion, consider a multi-language application, which consists of two files. The first file (Listing 11) contains JavaScript code and the second file (Listing 12) contains C code. The example allocates a JavaScript object (the `counter` object, see Listing 11), which is then used by the C code (see Listing 12). The JavaScript code exports the object `counter` using the built-in function `export`, stores the reference to the JavaScript object into the

[22] *Mapping of messages to language-specific operations and vice versa*, Grimmer, 2017: http://ssw.jku.at/General/Staff/Grimmer/TruffleVM_table.pdf.
[23] *Truffle JavaDoc*, Oracle, 2017: http://lafo.ssw.uni-linz.ac.at/javadoc/truffle/latest/.

```
var counter = {
    counterValue: 0,
    add: function(val) {
        counterValue += val;
    },
    myPrint: function() {
        print(counterValue);
    }
}
Truffle.export("counter", counter);
```

Listing 11. Allocation of a JavaScript object.

```
#include <truffle.h>

typedef struct {
    void (*add)(void *this, int val);
    void (*myPrint)(void *this);
} Counter;

int main() {
    Counter *c = (Counter*) Import("counter");
    c->add(c, 42);
    c->myPrint(c);
}
```

Listing 12. C type definition for the foreign object and an object-oriented access operation.

multi-language scope of the application. On the opposite side, the C code imports the object using the C built-in function `import`, defined in `truffle.h`.

Ruby and JavaScript are more similar (both languages are dynamically typed, object-oriented, and use the automatic memory management of TruffleVM). To keep the example simple, we write this application in JavaScript and C. However, the following ideas also apply for Ruby.

Dynamic and Static Typing. Wrigstad et al. [59] describe a concept called *like types*, which allows integrating dynamically typed objects in statically typed languages. Dynamically typed objects can be used in a statically typed language by binding them to *like-type* variables. Operations on like-type variables are syntactically and semantically checked against the static type of these variables, but their actual validity is only checked at runtime. Our approach is similar, except that we bind foreign dynamically typed objects to pointer variables that are associated with static type information (e.g., `Counter*`). Whenever an operation is performed on such a pointer (e.g., a property access), we check dynamically whether this operation is valid on the foreign object and report an error otherwise.

Listing 12 shows a C program, which uses a JavaScript object `counter`. The C code associates the variable `c` with the static type `Counter*`. When the C code accesses the JavaScript object, we check dynamically whether `add` or `myPrint` exist and report an error otherwise.

Object-Oriented and Non-Object-Oriented Programming. Object-oriented languages allow programmers to create objects that contain both data and code, known as *fields* and *methods*. Also,

objects can extend each other using class-based or prototype-based inheritance. When accessing fields or methods, the object does a lookup and provides a field value or a method. *Generic access* allows us to retain the object-oriented semantics of an object access even if the host language is not object-oriented. Consider the `add` method invocation (from C to JavaScript) in Listing 12. TruffleC maps this access to the following messages:

$$\begin{aligned} &\text{CCall}(\text{CMemberRead}(t_c, t_{\text{add}}), t_c, t_{42}) \\ &\overset{f_C}{\longmapsto} \text{Execute}(\text{Read}(t_c, t_{\text{add}}), t_c, t_{42}) \end{aligned} \tag{13}$$

TruffleJS resolves this access to an AST snippet that does the lookup of method `add` and executes it:

$$\begin{aligned} &\text{Execute}(\text{Read}(t_c, t_{\text{add}}), t_c, t_{42}) \\ &\overset{g_{JS}}{\longmapsto} \text{JSCall}(\text{IsJS}(\text{JSReadProperty}(\text{IsJS}(t_c), t_{\text{add}})), t_c, t_{42}) \end{aligned} \tag{14}$$

A method call in an object-oriented language passes the `this` object (i.e., the receiver) as an implicit argument. Non-object oriented languages that invoke methods therefore need to explicitly pass the `this` object. For example, the JavaScript function `add` (see Listing 11) expects the `this` object as an implicit first argument. Hence, the first argument of the `add` method call in C is the `this` object `c`.

Vice versa, the signature of a non-object-oriented function needs to contain the `this` object argument if the caller is an object-oriented language. For example, if JavaScript calls a C function, JavaScript automatically passes the `this` object as the first argument. The signature of the C function therefore needs to add the `this` object as an explicit argument to its signature. This approach allows us to access object-oriented data from a non-object-oriented language and vice versa.

Our current approach does not feature cross-language inheritance, i.e., class-based inheritance or prototype-based inheritance is only possible with objects that originate from the same language.

Explicit and Automatic Memory Management. TLIs are running within TruffleVM and can exchange data, independent of whether the data is managed or unmanaged: TLIs keep unmanaged allocations on the native heap, which is not garbage collected. For example, TruffleC allocates data on the native heap. However, it represents all pointers to such data (i.e., pointers to values, arrays, structures, and functions) as managed Java objects of type `CAddress` that wrap a 64-bit native address [27] and attach type information to it. TruffleC uses this type of information to resolve a *generic access* message to a `CAddress` object in such a way that an AST snippet is returned which directly accesses the raw data stored on the native heap. The *generic access* thus allows accessing unmanaged data from a language that otherwise only uses managed data. The JavaScript and Ruby implementations allocate objects on the Java heap. If an application binds a managed object to a C variable, TruffleC keeps this variable as a Java object of type `Object`. Thus, the Java garbage collector can trace managed objects even if they are referenced from unmanaged languages.

If a C pointer variable references an object of a managed language, operations are restricted. First, pointer arithmetic on foreign objects is only allowed as an alternative to array indexing. For example, C programmers can access a JavaScript array either with indexing (e.g., `jsArray[1]`) or by pointer arithmetic (`*(jsArray + 1)`). However, it is not allowed to manipulate a pointer variable that is bound to a managed object in any other way (e.g. `jsArray = jsArray + 1`). Second, C pointer variables that are bound to managed objects cannot be casted to primitive values such as `long` or `int`. References to the Java heap cannot be represented as primitive values like it is possible for raw memory addresses. Finally, unmanaged data structures cannot store references to managed objects. For example, it is not possible to assign a reference to a managed JavaScript object to a C array of pointers. In other words, `Counter *array[] = {jsReference};` is forbidden.

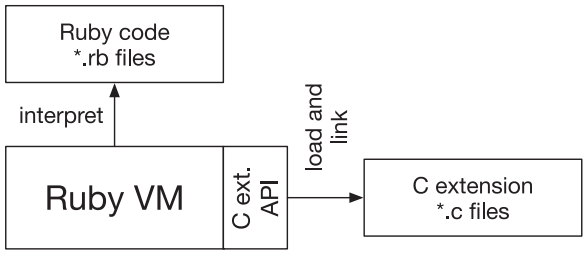

Fig. 9. Architecture of a Ruby application using C extensions.

Non-primitive C data is stored as a sequence of bytes on the native heap and it is not possible to keep managed references on the native heap. We report a runtime error in such cases.

Safe and Unsafe Memory Accesses. C is an unsafe language and does not check memory accesses at runtime, i.e., there are no runtime checks that ensure that pointers are only dereferenced if they point to a valid memory region and that pointers are not used after the referenced object has been deallocated. TruffleC allocates data on the native heap and uses raw memory operations to access it, which is unsafe. This has the following implications on multi-language applications:

If C shares data with a safe language, all access operations are *unsafe*. For example, accessing a C array from a JavaScript program is unsafe. If the index is out of bounds, the access has an undefined behavior (as defined by the C specification). On the other hand, accessing a C array is more efficient than accessing a dynamic JavaScript array because less runtime checks are required.

Accessing data structures of a safe language (such as a JavaScript array) from C is *safe*. TruffleC implements the access by a *Read* or *Write* message, which TruffleJS resolves with operations that check if the index is within the array bounds and grow the array in case the access was out of bounds.

TruffleVM, including the TLIs for JavaScript, Ruby, and C, eases an efficient multi-language development, which we evaluated by writing and executing multi-language benchmarks (see Section 5). We modified single-language benchmarks, which are available in C, Ruby, and JavaScript, such that parts of them were written in a different language. The only extra code that we needed was for importing and exporting objects from and to the multi-language scope.

4.2 C Extensions Support for TruffleRuby

In this second case study, we implemented the C extensions API for Ruby. A C extension is a C program that can access the data and metadata of a Ruby program by using a set of API functions (*C extension functions*), which are part of the Ruby VM MRI. Developers of a C extension for Ruby get access to this API by including the `ruby.h` header file. The C extension code is then dynamically loaded and linked into the Ruby VM when the Ruby program starts to run. Figure 9 gives an architectural overview of a Ruby application using a C extension.

We provide the same C extensions API as the Ruby MRI does, i.e., we provide all functions that are declared in `ruby.h`. To do so, we created our own implementation of `ruby.h`, which contains the function signatures of all the C extension functions. Listing 13 shows an excerpt of this header file including a description of the functions' semantics. In the following, we discuss how we can provide an implementation for these functions. We distinguish between *local* and *global* functions in the C extensions API. Local functions access and manipulate Ruby objects from within C. Global functions manipulate the global object of a Ruby application from C or directly access the Ruby engine.

```
typedef VALUE void*;
typedef ID void*;

// Define a C function as a Ruby method
void rb_define_method(VALUE class, const char* name,
                      VALUE(*func)(), int argc);

// Store an array element into a Ruby array
void rb_ary_store(VALUE ary, long idx, VALUE val);

// Get the Ruby internal representation of an identifier
ID rb_intern(const char* name);

// Get instance variables of a Ruby object
VALUE rb_iv_get(VALUE object, const char* iv_name)

// Invoke a Ruby method from C
VALUE rb_funcall(VALUE receiver, ID method_id, int argc, ...);

// Convert a Ruby Fixnum to a C long
long FIX2INT(VALUE value);
```

Listing 13. Excerpt of the `ruby.h` implementation.

4.2.1 Local Functions. TruffleC has knowledge about the C extension functions and substitutes every call to a local C extension function with a message in the AST that accesses the foreign Ruby data (we are using the same set of messages as in our first case study; see Section 4.1). The result is an AST $t' \in T_{N^{C \cup N^{\text{Msg}}}}$, which uses messages to access the foreign object rather than calling a function of the C extensions API.

TruffleVM can resolve these messages because TruffleRuby provides a mapping g_{Rb}. Message resolution uses g_{Rb} to map the messages in $t' \in T_{N^{C \cup N^{\text{Msg}}}}$ to an AST with a Ruby-specific access $t'' \in T_{N^{C \cup N^{\text{Rb}}}}$:

$$T_{N^{C \cup N^{\text{Msg}}}} \xrightarrow{g_{\text{Rb}}} T_{N^{C \cup N^{\text{Rb}}}} \tag{15}$$

The function to resolve the messages to Ruby-specific access operations (g_{Rb}) remained unchanged and we were able to reuse the infrastructure that was already implemented in TruffleRuby.

The following examples explain how we substitute the C extension functions `rb_ary_store`, `rb_funcall`, and `FIX2INT`.

- `rb_ary_store` allows writing an element of a Ruby array. Figure 10 shows how TruffleC substitutes the call of `rb_ary_store` (see source code in Listing 14) with a *Write* message for setting the Ruby array element. Upon first execution, this message is resolved by TruffleVM, which results in a TruffleC AST that does a Ruby array access via a setter function (`[]=`). The resolved AST replaces the AST for the *Write* message and is executed from now on.
- `rb_funcall` allows invoking a method on a Ruby object from within a C extension. TruffleC substitutes this call with two messages, namely, a *Read* message to get the method from the Ruby receiver and an *Execute* message, which invokes the method.
- `FIX2INT` transforms a Ruby Fixnum object to a C integer value. TruffleC substitutes this call with an *Unbox* message to the Ruby object.

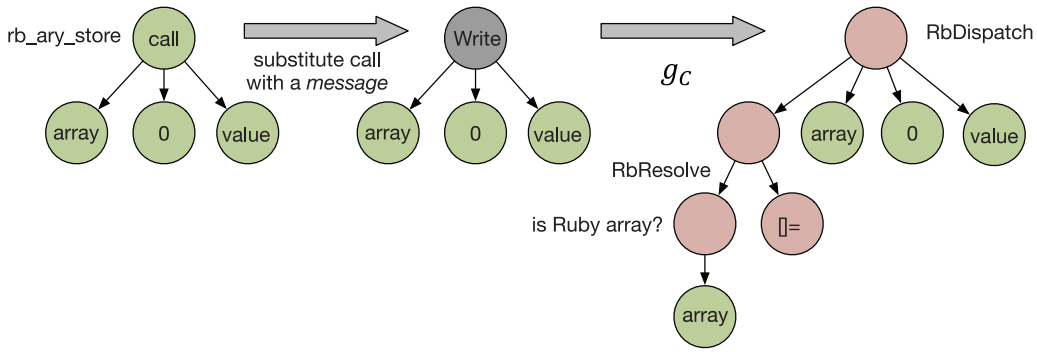

Fig. 10. TruffleC substitutes invocations of C extension functions with messages; TruffleVM resolves them to Ruby-specific operations.

```
1 #include <ruby.h>
2
3 VALUE array = ... ; // Ruby array of Fixnums
4 VALUE value = ... ; // Ruby Fixnum
5
6 rb_ary_store(array, 0, value);
```

Listing 14. Accessing a Ruby array from C.

4.2.2 Global Functions. The C extensions API also offers functions that manipulate the global object class of a Ruby application from C. For example, these functions can define global variables, modules, or functions. Global functions can also directly access the Ruby engine (e.g., to convert a C string to an immutable Ruby object). TruffleC forwards invocations of these global C extension functions to functions of the TruffleRuby engine.

In the following, we discuss how TruffleC implements calls to `rb_define_method` and `rb_intern`.

- `rb_define_method` allows defining a new method in a Ruby class. To substitute an invocation to this function, TruffleC directly accesses the Ruby engine and adds a C function pointer to a Ruby class object. The function pointer `VALUE(*func) ()` is a `CAddress` object, which references a TruffleC AST. When TruffleRuby invokes this method later, it uses *generic access*, i.e., it replaces a regular Ruby call with an *Execute* message. This message is then resolved to a TruffleC function call upon first execution.
- `rb_intern` provides a shared immutable Ruby object representation for a C string. TruffleRuby exposes a utility function that allows resolving these immutable Ruby strings, which TruffleC uses to substitute invocations of this methods.

4.2.3 Pointers to Ruby Objects. A pointer to a Ruby object is modeled in TruffleC as a Java reference to a `TruffleObject`. If the C program introduces additional indirection by applying the address-of operator (`&`) to a Ruby object reference, TruffleC creates an `MAddress` object (which itself extends TruffleObject; M stands for *managed*). `MAddress` objects wrap the foreign pointee, i.e., TruffleC uses `MAddress` objects to represent pointers to foreign objects. These objects allow achieving arbitrary levels of indirection. However, C also allows pointer arithmetic. This is frequently used in Ruby C extensions, as the API allows a pointer to be obtained to internal data

structures such as the C array that backs a Ruby string or array. It is then common to iterate directly over these arrays using pointers rather than Ruby API functions, in order to achieve higher performance. To support pointer arithmetic in TruffleC, an `MAddress` also holds an offset from the address of the pointee (i.e., the Ruby object). Pointer arithmetic just modifies this offset. Any dereferencing of this `MAddress` will use the same messages to read or write from the object as a normal array access would.

4.2.4 Discussion. We claim that TruffleVM eases an efficient multi-language development that also supports legacy interfaces between languages. In this case study, we presented an implementation of the C extensions API using *generic access*. We provided an implementation that allowed us to run Ruby code with C extensions that had been developed for real business applications (see also Section 5). We were able to successfully execute the existing modules `chunky_png`[24] and `psd.rb`[25], which are both open source and freely available on the RubyGems website. `chunky_png` is a module for reading and writing image files using the Portable Network Graphics (PNG) format. It includes routines for resampling, PNG encoding and decoding, color channel manipulation, and image composition. `psd.rb` is a module for reading and writing image files using the Adobe Photoshop format. It includes routines for color space conversion, clipping, layer masking, implementations of Photoshop's color blend modes, and some other utilities. Running the C extensions of these gems on top of TruffleVM required the following modifications for compatibility: TruffleC does not support variable-length arrays, hence, we replaced two instances of variable size stack allocations with a heap allocation via `malloc` and `free`. Running the C extensions on TruffleC also allowed us to find two bugs. A value of type `VALUE` (64-bit pointer value) was stored in a variable of type `int` (32-bit integer value), which caused different results between the Ruby module and the C extensions on all Ruby implementations. We have reported this implementation bug to the module's authors.[26] Apart from these minor modifications we are running all native routines from the two non-trivial gems unmodified.

5 PERFORMANCE EVALUATION

This section presents a performance evaluation of individual parts of TruffleVM. We present benchmarks that combine different languages and show that combining a slower language with a faster one yields an overall performance that is somewhere in the middle. We show that the C extensions API implementation with *generic access* runs benchmarks faster than all other Ruby implementations using C extensions. We also demonstrate that message resolution and cross-language inlining are essential for the performance of an application by measuring the effect of temporarily disabling them. We evaluate the composition of languages on top of TruffleVM with two different performance measurements.

First, we want to evaluate the performance of multi-language applications. Every language implementation can define efficient data representations, which can be shared across different languages. *Generic access* ensures that a TLI can directly access foreign objects. We expect using foreign data that are implemented in a TLI with slow access to have a negative effect on performances. On the other hand, we expect using foreign data that are implemented in a TLI with fast access to have a positive effect on performance. Accessing a Ruby array in C is less efficient than accessing a C array because Ruby arrays require more runtime checks. Accessing a C array in Ruby, however, is more efficient because a C array access performs a raw memory access without runtime checks. Message resolution inserts foreign-language-specific access operations into the AST of a host

[24] *Chunky PNG*, Willem van Bergen and others, 2015: https://github.com/wvanbergen/chunky_png.
[25] *PSD.rb from Layer Vault*, Ryan LeFevre, Kelly Sutton and others, 2015: https://cosmos.layervault.com/psdrb.html.
[26] *PSDNative*, Bug report, 2015: https://github.com/layervault/psd_native/pull/4.

application. However, it also inserts a language and type check before the foreign object is accessed. The dynamic compiler can often eliminate this additional check, which is discussed in more detail later. Furthermore, we claim that inlining across language boundaries as well as cross-language optimizations are critical for performance. We evaluate this claim by disabling message resolution. When disabling message resolution, L_{Host} still uses *generic access* to access foreign objects, but TruffleVM does not replace the messages in $t' \in T_{N^A \cup N^{Msg}}$. Rather, it uses $L_{Foreign}$ to locally execute the access operation and return the result. In other words, L_{Host} treats $L_{Foreign}$ as a black box, which introduces a language boundary.

Second, we compare the performance of TruffleVM to the C extensions API implementations of MRI, Rubinius, and JRuby. We claim that TruffleVM can run C extension functions on average over two times faster than natively compiled C code using MRI's C extensions API. In our evaluation, we back this claim and run image processing libraries that are implemented as C extensions. Also, we state that cross-language inlining and cross-language optimizations are the factors from which TruffleRuby benefits most compared to other Ruby implementations. Hence, we disable message resolution to verify this claim.

5.1 Evaluation Methodology

To account for the adaptive compilation techniques of Truffle and Graal, we set up a harness that executes each benchmark 50 times. After these warm-up iterations, every benchmark reaches a steady state such that subsequent iterations measure the fully optimized program and are identically and independently distributed. This was verified informally using lag plots [34]. We then sampled 10 iterations and calculated the averages for each configuration using the arithmetic mean. Where we report an error, we show the standard deviation. Where we summarize across different benchmarks, we report a geometric mean [17]. Our harness reports scores for each benchmark and its configurations. The score is the proportion of the execution count of the benchmark and the time needed (executions per second). We ran the multi-language benchmarks on an Intel Core i7-4770 quad-core 3.4GHz CPU running 64-bit Debian 7 (Linux3.2.0-4-amd64) with 16GB of memory. The C extensions benchmarks are long-running applications and the measurement takes several days. Therefore, we ran these benchmarks on a different hardware. We used a server machine with 2 Intel Xeon E5345 processors with four cores each at 2.33GHz and 64GB of RAM, running 64-bit Ubuntu Linux 14.04.

We focus this evaluation on peak performance of long-running applications where the start-up performance plays a minor role. Hence, we consider start-up performance to be out of scope and present performance numbers after an initial warm-up. A detailed evaluation of the start-up performance of Truffle can be found in Ref. [40].

5.2 Truffle Language Implementation Composition

In this section, we evaluate the peak performance of multi-language applications running on top of TruffleVM.

5.2.1 Interoperability between JavaScript, Ruby, and C. For this evaluation we used benchmarks that heavily access objects and arrays. The benchmarks (the SciMark benchmarks[27] and benchmarks from the Computer Language Benchmarks Game[28]) compute a Fast Fourier Transformation (FFT), a Jacobi successive over-relaxation (SOR), a Monte Carlo integration (MC), a sparse matrix multiplication (SM), a dense LU matrix factorization (LU), sort an array using a tree data structure

[27] *SciMark 2.0*, Roldan Pozo and Bruce R Miller, 2015: http://math.nist.gov/scimark2/index.html.
[28] *The Computer Language Benchmarks Game*, Brent Fulgham and Isaac Gouy, 2015: http://benchmarksgame.alioth.debian.org/.

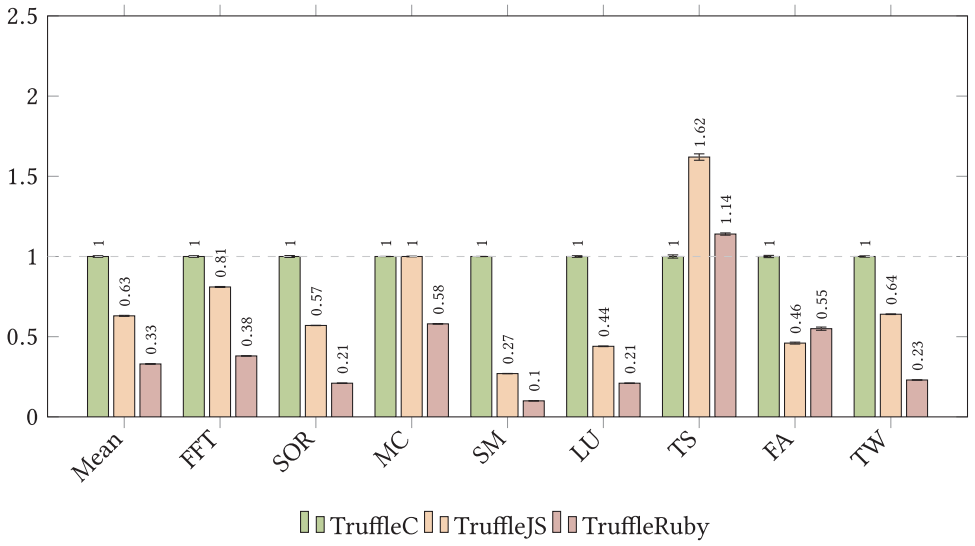

Fig. 11. Performance of individual languages on our benchmarks (normalized to C performance; higher is better).

(TS), generate and write random DNA sequences (FA), and solve the towers of Hanoi problem (TW).

First, we compare the performance of the individual TLIs on our benchmarks. For that, we translated the C version of our benchmarks to JavaScript and Ruby. Having the exact same benchmark implemented in different languages guarantees a fair performance comparison. The results in Figure 11 are normalized to the TruffleC performance.

Second, we ran the multi-language version of our benchmarks. We modified every C, JavaScript, and Ruby benchmark such that all array and object allocations were extracted into factory functions. We then implemented these factory functions in the other two languages arriving at multi-language benchmarks in which C, JavaScript, or Ruby programs call factory functions in C, JavaScript, or Ruby (in any combination). For example, a C program whose allocation factory functions are written in JavaScript works with JavaScript objects. The SciMark benchmarks (FFT, SOR, MC, SM, LU) already had factory functions for all allocations and it was easy to replace these functions with versions that are written in different languages. For TS, we implemented the allocation of the tree data structure in different languages. The FA benchmark was modified in a sense that the DNA data structures were allocated using different languages. Finally, the TW benchmark also allocates the tower data structures using different languages. We grouped our evaluations such that their main part was either written in C, in JavaScript, or in Ruby. For each group, we used the single-language implementation as the baseline (i.e., factory methods written in the same language as the main program) and show how multi-language applications perform compared to single-language applications. The x-axis of each chart in Figures 11, 12, 13, 14, and 16 shows the different benchmarks. The y-axis of each chart shows the average scores of the benchmarks (higher is better). We based TruffleVM on Graal revision `bf586af6fa0c`.

Results of Single-Language Benchmarks. Figure 11 shows that JavaScript code is on average 37% slower and Ruby code is on average 67% slower than C code. C is efficient because C data accesses do not require runtime checks (such as array bounds checks), but the memory is accessed

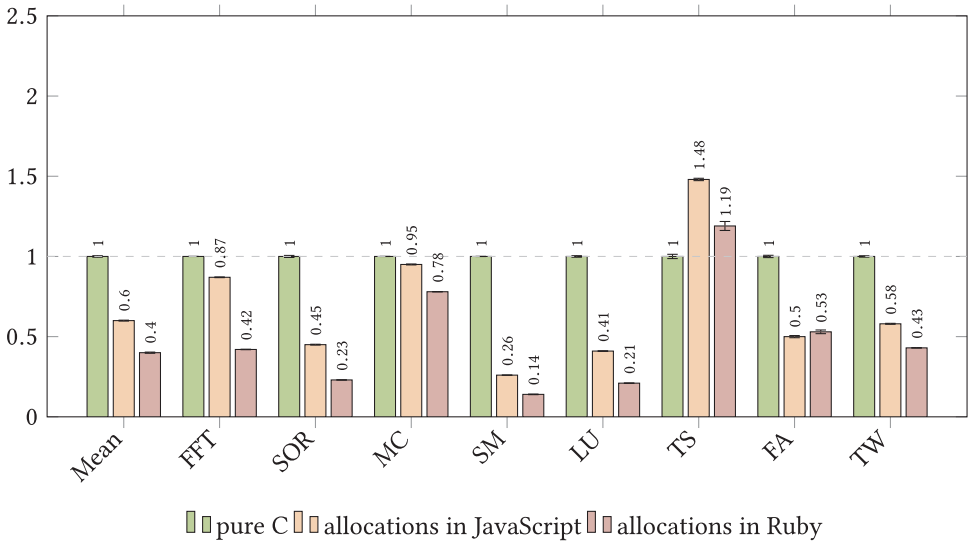

Fig. 12. Main part in C; allocations in different languages (normalized to pure TruffleC performance; higher is better).

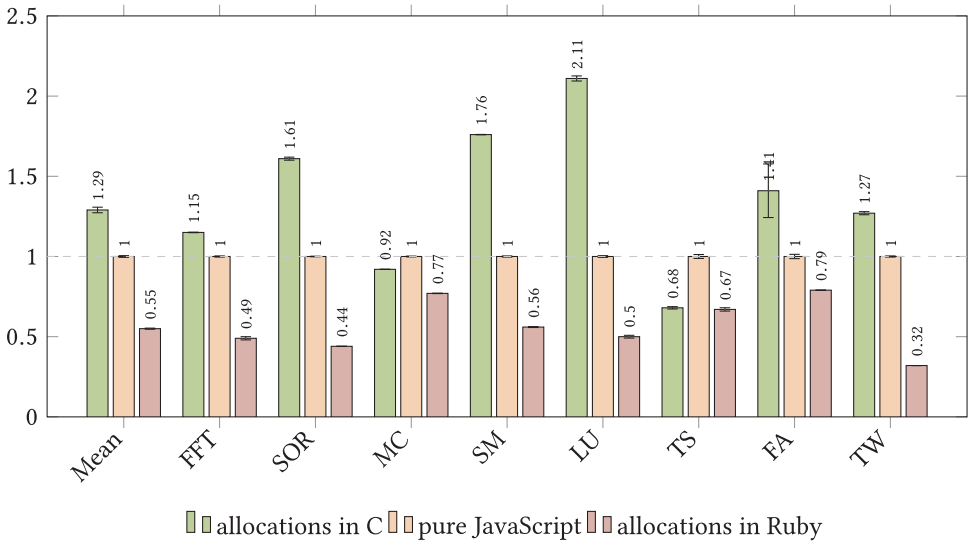

Fig. 13. Main part in JavaScript; allocations in different languages (normalized to pure TruffleJS performance; higher is better).

directly. This efficient data access makes C the fastest language for most benchmarks. However, if a program allocates data in a frequently executed part of the program, the managed languages (JavaScript and Ruby) can outperform C. Allocations in TruffleC (using `calloc`) are more expensive than the instantiation of a new object on the Java heap. TruffleC does a native call to execute the `calloc` function of the underlying OS. TruffleJS or TruffleRuby allocate a new object on the Java heap using sequential allocation in thread-local allocation buffers, which explains why JavaScript

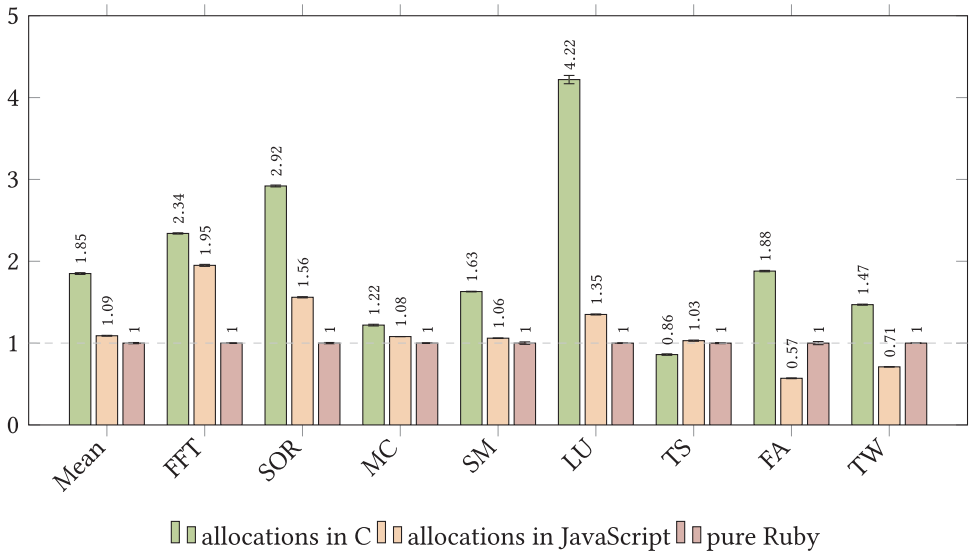

Fig. 14. Main part in Ruby; allocations in different languages (normalized to pure TruffleRuby performance; higher is better).

and Ruby perform better than C on TS. This benchmark allocates data in a hot loop. The Ruby semantics require that Ruby objects are accessed via getter or setter methods. TruffleRuby uses a dispatch mechanism to access these methods. This dispatch mechanism introduces additional runtime checks and indirections, which explains why Ruby is in general slower than JavaScript or C.

Results of Multi-Language Benchmarks. The multi-language versions of our benchmarks heavily access foreign objects:

- *C Objects:* C data structures are unsafe; access operations are not checked at runtime, which makes them efficient in terms of performance. Hence, using C data structures from JavaScript or Ruby applications improves the runtime performance. However, an allocation with `calloc` is more expensive than an allocation on the Java heap. Thus, factory functions in JavaScript or Ruby perform better than factory functions written in C.
- *JS Objects:* TruffleJS uses an object implementation where each access involves runtime checks. Examples of such checks are array bounds checks to dynamically grow JavaScript arrays or property access checks to dynamically add properties to an object. These checks are the reason why accesses to JavaScript objects perform worse than accesses to C objects.
- *Ruby Objects:* TruffleRuby's dispatch mechanism for accessing objects introduces a performance overhead compared to JavaScript and even more so to C. According to the Ruby semantics, TruffleRuby invokes getter and setter methods to access the data. This additional indirection is the reason why accesses to Ruby objects are in general slower than accesses to JavaScript objects or C objects.

We can show that the performance of a multi-language program mainly depends on the performance of the individual language parts. Using JavaScript or Ruby data from C programs has a negative impact on the overall performance. Figures 12 and 13 show that using Ruby objects from C or JavaScript programs causes an overhead of up to 86%. On average, using Ruby data from

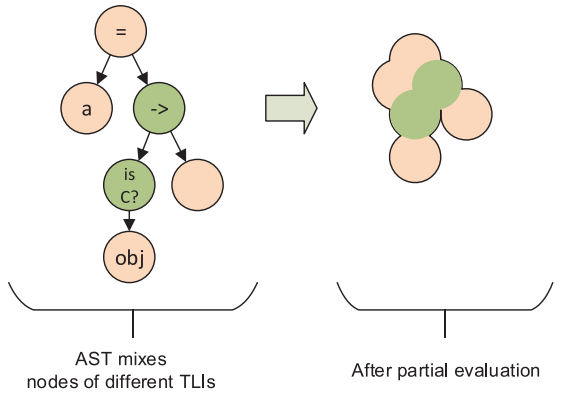

Fig. 15. Language boundaries are completely transparent to the compiler.

C causes a performance overhead of 60% and using Ruby data from JavaScript causes a performance overhead of 45%. On the other hand, using efficient foreign data has a positive effect on performance. For example, Figures 13 and 14 show that using efficient C data from JavaScript or Ruby programs can improve performance by up to 322%. On average, using C data improves the JavaScript performance by 29% and the Ruby performance by 85%.

TruffleVM does not marshal objects at the language boundary but directly passes them from one language to another and the TLIs use *generic access* to access them. Message resolution only affects the application's performance upon the first execution of an object access. Message resolution integrates AST snippets from different languages and thus creates a uniform AST (see Figure 15) that allows the Graal compiler to apply its optimizations across language boundaries (e.g., *cross-language inlining*). Widening the compilation unit across different languages is important [2, 49] as it enables the compiler to apply optimizations to a wider range of code, e.g., it allows the compiler to apply *escape analysis* and *scalar replacement* [51] to foreign objects. Consider a JavaScript program that allocates an object, which is used by a C part of the application. Message resolution ensures that Graal's escape analysis can analyze the object accesses, independent of the host language. If the JavaScript object does not escape the compilation scope, scalar replacement can remove the allocation and replace all usages of the object with scalar values. To demonstrate the performance improvement caused by message resolution, we temporarily disabled it. In Figure 16, we show the performance of our JavaScript benchmarks using C data structures with and without message resolution. When disabling message resolution, every data access as well as every function call crosses the language boundary, which results in a performance overhead of more than 500%. We expect similar results for the other configurations, but we have not measured them because disabling message resolution for a TLI requires a significant engineering effort.

The dynamic compiler can also minimize the effect of *generic access*'s language and type check on the receiver. Using conditional elimination [50], the Graal compiler can even remove the additional type/language check by merging it with the type checks on the receiver, which is necessary in dynamically typed languages anyway. Conditional elimination reduces the number of conditional expressions by performing a control-flow analysis over the IR graph and pruning conditions that can be proven to be true. Also, it can move checks resulting from *generic access* out of loops if the compiler can prove that a condition is loop-invariant [13, 14].

5.2.2 C Extensions Support for Ruby. In this section, we compare TruffleVM to other multi-language systems that can compose Ruby code and native C code, namely MRI, Rubinius, and

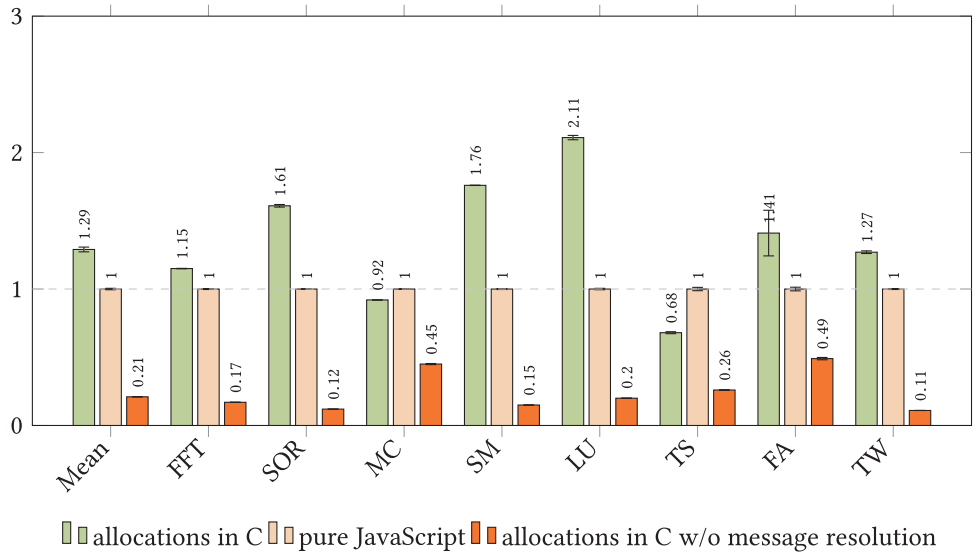

Fig. 16. Main part in JavaScript and allocations in C with and without message resolution (normalized to pure TruffleJS performance; higher is better).

JRuby. We show that TruffleVM performs better than these related approaches. We benchmarked examples of real-world C extensions that were developed to meet a real business need (see also Section 4.2.4). Also, we used code that is computationally intensive rather than I/O intensive, as our system does nothing to improve I/O performance. To the best of our knowledge, there is no widely accepted benchmark suite for evaluating the performance of native C extensions. Therefore, we used the existing Ruby modules `chunky_png` and `psd.rb`. Both modules have separately available C extension modules, namely `Oily_png`[29], which includes C extensions for resampling, PNG encoding and decoding, color channel manipulation, and image composition as well as `Psd-native`[30], which contains C extensions for color space conversion, clipping, layer masking, implementations of Photoshop's color blend modes, and some other utilities. These algorithms are available to Ruby programmers as a library. We evaluated all available 43 image processing functions, each providing a different algorithm to manipulate image data. The 43 routines were set up in a benchmark harness, which allocates Ruby data that is then processed in a C extension, i.e., the computations of these routines are implemented in C but the data is provided by Ruby. This harness simulates a Ruby developer applying different manipulations to image data, i.e., we measure the runtime of the individual algorithms to process Ruby image data.

We based TruffleVM on Graal revision `9535eccd2a11`. Where an unmodified Java VM was required, we used the 64-bit JDK 1.8.0u5 with default settings. Native versions of Ruby and C extensions were compiled with the system standard GCC 4.8.2. Figure 17 summarizes the peak performances of all 43 C extensions benchmarks by showing their geometric mean speedup (y-axis, higher is better). We compare all implementations (x-axis) relative to the speed at which the C extensions run when using MRI's implementation of the C extensions API. A detailed table with performance numbers for all 43 C extension benchmarks can be found in Ref. [26].

[29] *OilyPNG*, Willem van Bergen and others, 2015: https://github.com/wvanbergen/oily_png.
[30] *PSDNative*, Ryan LeFevre, 2015: https://github.com/layervault/psd_native.

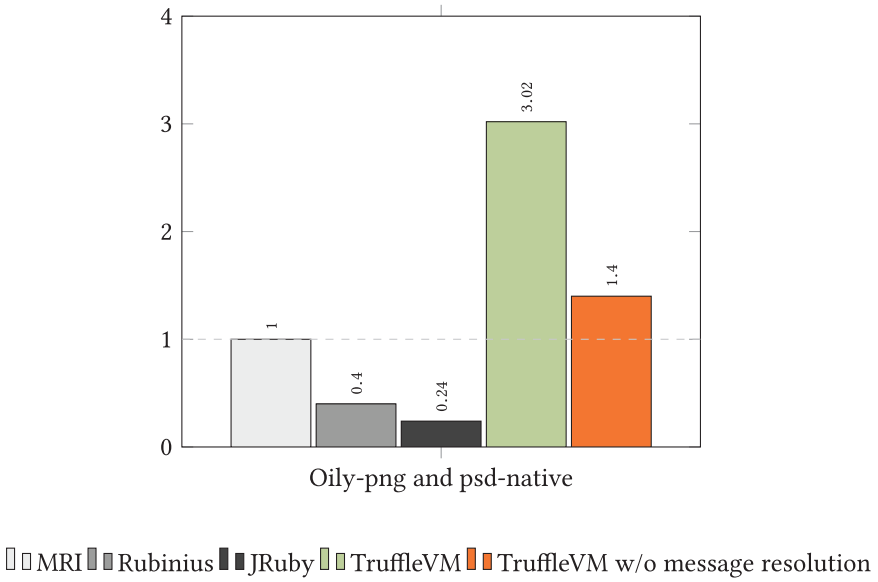

Fig. 17. C extensions benchmarks (normalized to natively compiled C extensions that interface to MRI; higher is better).

The standard implementation of Ruby is known as MRI, or CRuby. It is a bytecode interpreter with some simple optimizations such as inline caches for method dispatch. MRI has excellent support for C extensions, as the API directly interfaces with the internal data structures of MRI. We evaluated version 2.1.2.

Rubinius is an alternative implementation of Ruby using a VM core written in C++ and using LLVM to implement a simple JIT compiler, but much of the Ruby-specific functionality in Rubinius is implemented in Ruby. Rubinius uses internal data structures and implementation techniques different from those in MRI. Most importantly, it uses C++ instead of C; so to implement the C extensions API, Rubinius has a bridging layer, which converts C extensions API calls to C++ calls of the Rubinius implementation. We evaluated version 2.2.10.

JRuby is an implementation of Ruby on the JVM. It uses dynamic class file generation and the `invokedynamic` instruction to JIT-compile Ruby to JVM bytecode, and thus to machine code. JRuby uses Java's JNI [38] to implement a bridging layer that supports MRI's C extensions API. This technique is almost the same as in Rubinius, except that now the interface between the VM and the bridging layer is even more complex. To share Ruby data with C extensions, JRuby must copy the data from the managed Java heap onto the unmanaged native heap. Whenever the native data is modified, JRuby copies the changes back to the managed Ruby object. To keep both sides synchronized, JRuby must do this copying whenever data is passed between Ruby and C or vice versa. JRuby had some experimental support for C extensions, but after initial development it became unmaintained and has since been removed. We evaluated the last major version where we found that the code still worked, which was version 1.6.0.

TruffleVM is our system. We implemented the C extensions API with *generic access*. As before (see Section 5.2.1), Figure 17 also shows performance numbers for a configuration that disables message resolution.

As the baseline of our evaluation, we used natively compiled C extensions that interface to MRI. C extensions that interface to Rubinius are on average 60% slower than the baseline. Rubinius

needs a bridging layer to meet MRI's API, which introduces a significant runtime overhead. Three of the benchmarks also failed to make any progress in this configuration, so we considered them to have timed out and did not include their runtimes in the reported mean value. C extensions that interface to JRuby are on average 76% slower than C extensions that interface to MRI. JRuby has a JNI-based bridging layer to meet MRI's API, which causes a significant overhead. One benchmark failed with an error about a missing feature; 17 benchmarks did not make progress in reasonable time. C extensions that run on top of TruffleVM are on average 202% faster than the baseline. TruffleVM performs better because MRI, Rubinius, and JRuby run the Ruby code in a dedicated VM; only the C extensions are statically compiled and run natively. Every call or data access from native code to the VM (and vice versa) is a compilation barrier that prevents the compiler from performing any optimizations across the language boundaries. However, in our system, the C extensions are executed on top of TruffleC and are therefore running in the same VM as the Ruby code. We use *generic access* for any foreign object access (C extensions accessing Ruby data), which removes the language boundaries completely and allows optimizations across languages. Our system performs best on C extensions that heavily access Ruby data but otherwise do little computation. *Generic access* removes all language boundaries, which in the best case allows compiling the entire benchmark into a single machine code routine. Performance is similar to native C extensions that interface to MRI if the benchmarks are computationally intensive. In these cases, the performance numbers are dominated by the computationally intensive parts rather than by the foreign data access. When disabling message resolution, the C extensions run 54% slower than with message resolution. However, their performance is still 40% faster than native C extensions that interface to MRI.

5.3 Discussion

In this performance evaluation, we showed that if a TLI accesses foreign data using *generic access*, the performance of this access mainly depends on the implementation of the data structure and its access operations. We demonstrated that the Graal compiler removes most of the language and type checks in *generic access*. The performance of a multi-language program depends on the performance of the individual language parts. Using foreign data of a slow language implementation has (as expected) a negative impact on performance (e.g., Ruby objects used in C). On the other hand, using foreign data implemented in an efficient language has a positive effect on performance (e.g., C data used in Ruby).

Generic access yields excellent performance of multi-language applications because of two reasons: First, message resolution replaces language-agnostic messages with efficient foreign-language-specific operations. Accessing foreign objects becomes as efficient as accessing objects of the host language. Second, the dynamic compiler can perform optimizations across language borders because these borders were removed by message resolution. For example, the compiler can inline C functions into Ruby code and vice versa, which enables optimizations across language boundaries.

6 LESSONS LEARNED

The work described in this article allowed us to identify three key aspects for efficient and extensible cross-language interoperability.

(1) *Multi-language programming model:* The first step is to define a programming model that allows combining different languages. In this work, we used two different approaches: First, we introduced a novel multi-language programming model that combines languages via a multi-language scope that programmers can use to exchange data and functions

Table 1. Code Metrics of TruffleVM

	Project Size (Lines of Code)	Modification Size (Lines of Code)	Modification Description
Dynamic Compiler: Graal	240k		No changes necessary
Truffle Framework	84k	3k	Message resolution API Multi-language scope
TruffleJS	127k	5.4k	Message resolution implementation
TruffleRuby	129k	3.7k	Message resolution implementation C Extension API implementation
TruffleC	34k	3.4k	Message resolution implementation

across languages. Multi-language applications can access foreign objects and can call foreign functions by simply using the operators of the host language. Second, we implemented an existing FFI and used *generic access* to implement the interface.

(2) *Language-agnostic foreign object access:* We conclude that a language-agnostic set of primitive cross-language operations (we call them *messages* as part of *generic access*) makes our approach flexible enough to support different interoperability use-cases. We believe that a language-agnostic set of operations is important as it allows adding new languages without requiring modifications in the existing language implementations. If the set of operations were specific to a combination of languages, any new language would require modification of existing language implementations as part of TruffleVM (the engineering effort would be quadratic with the number of languages).

Also, pushing the abstraction of object accesses down to primitive operations (e.g., read and write messages) features enough flexibility to meet the needs of a variety of languages (e.g., raw memory accesses in C and an object accesses in JS) and use-cases (e.g., writing multi language applications in three different languages as shown in Section 4.1 or using an FFI as shown in Section 4.2).

(3) *Optimization across language boundaries:* The key insight in terms of performance is that a uniform approach of language implementations on the same VM is an essential factor. We used Truffle ASTs as a common intermediate representation of our languages. Message resolution as part of *generic access* allows us to compose ASTs of different languages and therefore to remove the boundaries between languages. These unified ASTs allow us to use an unmodified dynamic compiler for optimization across language boundaries, which ensures efficient cross-language interoperability.

To round out our experience and lessons learned, we describe the implementation effort of all components as part of TruffleVM, the limitations, as well as future work and research enabled by this work.

6.1 Implementation Effort

We based our work on Truffle and Graal and extended the Truffle framework by *generic access*. Table 1 shows the sizes of the various TruffleVM parts as well as the necessary modification efforts. TruffleJS is an Oracle Labs project, which was started in 2012 as a student project at Johannes Kepler University. As of March 2017, TruffleJS is being developed by a team of seven people. In total, we estimate the implementation effort of TruffleJS with 19 person-years. TruffleRuby is also an Oracle Labs project, which was started in 2014, and by the time this article is written, TruffleRuby will be in development by a team of six people. In total, we estimate the implementation effort of

TruffleRuby with 10 person-years. TruffleC is a research project developed at the Johannes Kepler University. It was implemented by two PhD students within two years.

For the work presented in this article, we extended the Truffle framework by an API for *generic access*. This API defines all messages and the necessary features for message resolution. Also, our extensions include the multi-language scope. In total, our extensions added 3k LOC to the Truffle framework. We were able to reuse the Graal compiler without any modifications because *generic access* produces unified ASTs of different languages, which can be directly compiled by Graal.

For the case studies of this article, we extended existing TLIs with *generic access*, namely TruffleJS, TruffleRuby, and TruffleC. These extensions add 5.4k LOC to TruffleJS, 3.7k LOC to TruffleRuby, and 3.4k LOC to TruffleC. Extending an existing TLI by adding support for *generic access* requires the following engineering effort:

— *Generic access - transforming foreign object accesses to language-agnostic messages:* If a TLI wants to act as a host language and to access objects of a foreign language, it needs to map these accesses to messages, i.e., a TLI (L_{New}) has to define $T_{N^{New}} \xrightarrow{f_{New}} T_{N^{New} \cup N^{Msg}}$. Optionally, a TLI can provide an API that allows programmers to explicitly use *generic access*.
— *Generic access - transforming language-agnostic messages to regular object accesses:* If a TLI (L_{New}) wants to be used as a foreign language and to share objects with other languages, shared objects need to support *generic access*. The TLI needs to define a mapping from language-agnostic messages to access operations that are specific to L_{New}: $T_{N^{A} \cup N^{Msg}} \xrightarrow{g_{New}} T_{N^{A} \cup N^{New}}$.
— *Multi-language scope:* The TLI has to provide infrastructure for the application programmer to export and import objects to and from the multi-language scope.

Extending Truffle by the API for *generic access* and extending three existing languages by an implementation of *generic access* could be accomplished by one Truffle expert within one year.

If one wants to add a new language to TruffleVM, it is first necessary to implement the guest language as a Truffle language implementation. The layered approach of Truffle simplifies guest language implementations. With Truffle, common parts found in every high-performance VM are factored out, which allows developers to focus on required execution semantics when implementing a TLI. Only guest-language-specific parts have to be implemented from scratch. A core of reusable host services is provided by the framework, such as dynamic compilation, automatic memory management, threads, synchronization primitives, and a well-defined memory model.

6.2 Limitations and Future Work

We identified the following limitations and future research opportunities in cross-language interoperability:

— *Limited support for precompiled libraries:* Our approach is based on a shared intermediate representation of source code. We compile code snippets to Truffle ASTs and combine these ASTs using *generic access*. This technique imposes the requirement that the source code of an application is available. All source code that needs to operate on foreign objects needs to be executed on top of a TLI.

  Nevertheless, individual languages can still link to precompiled code. For example, TruffleC can link to the standard library and access its function via GNFI (c.f. Section 2). However, foreign objects cannot be passed to this library code; e.g., it is not possible to pass a JavaScript object to the native `printf` function of the C standard library. In these cases, we report a runtime error.

— *Managed and unmanaged data:* As described in Section 4.1.2, when combining managed and unmanaged languages, operations are restricted. For example, unmanaged data structures cannot store references to managed objects.

However, we are working on a memory-safe version of our C implementation [24], which uses managed Java objects to represent native allocations. Using managed allocations to represent C data would allow us to overcome these limitations. Our work on a memory-safe C implementation and its combination with other languages is ongoing research and we are not yet able to provide results or an evaluation.

— *Multi-language concurrency, parallelization, and threads:* Truffle allows implementing different models for concurrency on top of TruffleVM. However, we consider multi-threaded applications that mix different languages and concurrency models a separate research topic and out of scope of this work. The work of this article is limited to single-threaded applications. The concurrency models in JavaScript, Ruby, and C differ significantly, i.e., applications written in multiple languages need to unify these models and bridge the differences across languages. Our ongoing research investigates concurrency and parallelization across language borders.

The work presented in this article is the basis of our ongoing research: Daloze et al. [12] investigates a tread-safe and language-agnostic object model. Also, there is ongoing work that explores an implicit parallel-data programming model combining JavaScript data structures with C functions.

— *Multi-language inheritance:* Currently, there is no support for cross-language inheritance, i.e., class-based inheritance or prototype-based inheritance is only possible with classes or objects that originate from the same language. However, we are convinced that TruffleVM is extensible in this respect. Therefore, future research could investigate inheritance across language boundaries.

— *Cross-language debuggers:* The Truffle framework allows the implementation of debuggers with zero-overhead [47]. Future research will focus on generalizing the existing debuggers for TLIs so that they can be used for multi-language applications; similar to the work of Lee et al. [37]. The goal of this work is a zero-overhead debugger for multi-language applications that allows developers to step into functions or inspect data, which were implemented or allocated in different languages.

— *Additional languages:* TruffleVM allows adding new TLIs easily. A TLI has to support *generic access* and needs access to the multi-language scope. Having these extensions, the TLI can be added to TruffleVM. As future work, we want to add further languages to the runtime. There is a TLI for the functional language Clojure, for the dynamic language Python[31] [57, 63], as well as for Java [21]. As we did for JavaScript, Ruby, and C (see Section 4.1.2), this work requires bridging different language paradigms and features. We did not include these languages into the case study of this article because at the time of writing this article, Clojure and Java were in an early state and under heavy development. Python was developed by external collaborators. Hence, adding these TLIs to TruffleVM would have required a major engineering effort and was therefore intentionally left for future work.

There is ongoing work that integrates the mathematical language R[32] into TruffleVM. However, this is early work and we are not yet able to provide results or an evaluation.

[31] *Zippy—a Python implementation on top of Truffle*, Bitbucket repository, 2015: https://bitbucket.org/ssllab/zippy.
[32] *FastR—an R implementation on top of Truffle*, Oracle, 2017: https://github.com/oracle/fastr.

— *C extensions support for other languages:* Besides Ruby, other dynamic languages also have a C extensions API, e.g., Python[33] or R.[34] Similar to our implementation for TruffleRuby, an implementation of the C extensions API could simply substitute invocations of these extension functions with a *generic access*. Also, TruffleJS already supports Node.js including native modules. However, these native modules are currently integrated using a JNI based implementation. As part of future work we want to run JS modules as well as native modules on TruffleVM and integrate them using *generic access* as proposed in this article.

7 RELATED WORK

To put TruffleVM in context, we compare it to related approaches for cross-language interoperability, including foreign function interfaces, inter-process communication, and multi-language runtimes.

7.1 Foreign Function Interfaces

Most modern VMs expose an FFI such as Java's JNI [39], Java's Native Access,[35] or Java's Compiled Native Interface.[36] An FFI defines a specific API between two languages. Programmers can access foreign objects of the target language by using this API. However, the result is rather inflexible: in order to interact with a foreign language, the programmer has to write glue code and this code only works for a specific pair of languages. Also, FFIs primarily allow integrating C/C++ code into languages such as Ruby (Ruby's C extensions mechanism), R (native R extensions), or Java [39]. They hardly allow integrating code written in a different language than C.

Wrapper generation tools (e.g., Refs. [6] and [45]) use annotations to generate FFI code from C/C++ interfaces, rather than requiring users to write FFI glue code by hand. A similar approach is described in Ref. [36], where existing interfaces are transcribed into a new notation instead of using annotations.

Compilation barriers at language boundaries have a negative impact on performance. To widen the compilation span across multiple languages, Stepanian et al. [52] describe an approach that allows inlining native functions into a Java application using a JIT compiler. They can show how inlining substantially reduces the overhead of JNI calls.

Kell et al. [35] describe *invisible VMs*, which allow a simple and low-overhead foreign function interfacing. They implement the Python language and minimize the FFI overhead to natively compiled code.

Jeannie [29] allows toggling between C and Java; hence, the two languages can be combined without writing boilerplate code. In Jeannie, programmers can mix both Java and C code in the same file and Jeannie compiles this to code parts communicating via JNI.

There are many other approaches that target a fixed pair of languages [8, 19, 33, 46, 59]. These approaches are all tailored toward interoperability between two specific languages and cannot be generalized for arbitrary languages and VMs. In contrast to them, our solution provides true cross-language interoperability between any Truffle-based languages rather than just pairwise interoperability. We can compose languages and reduce boilerplate code to a minimum. We do not target a fixed set of languages, and *generic access* does not introduce a compilation barrier when crossing language boundaries.

[33] *Python Language*, Python Software Foundation, 2015: https://www.python.org/.
[34] *The R Project for Statistical Computing*, The R Foundation, 2015: http://www.r-project.org/.
[35] *Java Native Access (JNA)*, GitHub repository, 2015: https://github.com/twall/jna.
[36] *Compiled Native Interface (CNI)*, GCC the GNU Compiler Collection, 2015: http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html.

7.2 Inter-Process Communication

IDLs implement cross-language interoperability via message-based inter-process communication between separate runtimes. This approach is mainly targeted to remote procedure calls and often not only aims at bridging different languages but also at calling code on remote computers. Programmers can define an interface in an IDL that can then be mapped to multiple languages. An IDL interface is translated to stubs in the host language and in the foreign language, which can then be used for cross-language communication [43, 44, 48, 55]. These per-language stubs marshal data to and from a common wire representation. However, this approach introduces a marshalling and copying overhead as well as an additional maintenance burden (learning and using an IDL, together with its toolchain).

Lightweight remote procedure calls [7] optimize the communication between protected domains on the same machine (including control transfer, data transfer, and linkage), which reduces the costs incurred by using remote procedure calls. Otherwise, using IDLs in the context of single-process applications has only been explored in limited ways [16, 56].

Generic access is also based on messages, but they are resolved at runtime and are replaced with direct access operations to foreign objects. These messages are transparent to the programmer and are automatically generated. TruffleVM makes the mapping of foreign language operations to messages the task of the language implementer rather than the task of the application programmer. Our approach accesses foreign objects directly instead of copying them at language borders. In fact, language borders are completely eliminated so that the dynamic compiler can optimize across languages and can thus improve the performance of multi-language applications significantly.

7.3 Multi-Language Runtimes

TruffleVM composes language implementations that are running on a shared VM, which is closely related to Microsoft's Common Language Runtime [10, 15, 42] as well as to the RPython [9] runtime. In the following, we compare these runtimes to TruffleVM.

Microsoft's CLI. The Microsoft Common Language Infrastructure (CLI) [15] describes language implementations that compile different languages to a common IR that is executed by the CLR [10]. The CLR provides a common type system, automatic memory management, a JIT compiler (a function is compiled just before execution), a security manager, and a class loader. The CLR can execute conventional object-oriented imperative languages, dynamically typed languages, and functional languages (e.g., F#).

The Dynamic Language Runtime (DLR) [28] is a framework for implementing dynamic languages on top of the CLR, which is similar to Truffle as a framework on top of the JVM. Language developers parse source code to an expression tree, which is the DLR's representation of source code. The DLR defines a fixed set of language-agnostic expressions that language implementers use to build up an expression tree. DLR expression trees can be interpreted by the DLR's interpreter or converted to the CLR's IR, which is directly compiled to machine code. A DLR language implementation transforms operations of dynamically typed operands to *call sites* [28]. For example, an implementation does not emit IR code that adds two numbers for a JavaScript + operation, but it emits a call site. A call site is a placeholder for an operation that is resolved at runtime. The DLR uses a *delegate* to implement a call site. A delegate then calls the different implementations of an operation. In the case of a JavaScript + operation, the delegate can call the `double` instance of a + operation (implemented as a type check followed by a `double` addition). This approach is different to Truffle because Truffle ASTs are self-optimizing and speculatively rewrite themselves with *specialized* variants at runtime, e.g., based on profile information. This technique allows specializing

on a subset of the semantics of a particular operation. Truffle compiles frequently executed ASTs to machine code and deoptimizes them if a tree needs to be *re-specialized*.

A language implementation on top of the DLR needs to use the object model of the CLR to implement the objects of a guest language, i.e., it has to represent them using CLR's builtin types [28]. DLR languages can make dynamic calls on objects defined in other languages. Similar to *generic access*, the DLR is inspired by Smalltalk. It defines a meta-object protocol that provides a set of language-agnostic operations on objects. These operations are again implemented with call sites and delegates.

Microsoft's approach is different from ours because of the following reasons:

— *Object representation:* Language implementations on top of the CLR (including the DLR languages) need to use the statically typed and managed object model of the CLR. Tight interoperability on the IR level is only possible between languages whose type system corresponds to the CTS. The object access is implemented using the CLR's IR.

  Generic access, on the other hand, allows every language to have its own representation of objects and to define individual access operations. TLIs are not bound to a common object representation, e.g., TruffleJS allocates managed objects whereas TruffleC allocates data as plain byte sequences on the unmanaged native heap. *Generic access* resolves and embeds language-specific AST snippets for each access at runtime (e.g., access operations to the managed Java heap or a raw memory access to the native heap).

— *Languages:* The CLR accesses unmanaged code (e.g., C code) via the annotation-based PInvoke and the FFI-like IJW interfaces, which use explicit marshalling and a pinning API. TruffleVM treats unmanaged languages (e.g., C) as first-class citizens and provides a TLI for them. TruffleC supports *generic access* and can access managed and dynamically typed objects. Also, other high-level languages (e.g., JavaScript) can access unmanaged C data efficiently.

— *Object access:* The DLR uses cached delegates to access foreign objects, which causes a call site for every foreign object access. Jeff Hardy states in Ref. [28] that a foreign object access is *as fast as other dynamic calls, and almost as fast as static calls. Generic access* avoids this indirection, i.e., there are no call sites between languages because *generic access* directly embeds the foreign object access into the AST of the host application. We specialize a foreign object access on the language and the type of the foreign object and embed it into the host language's AST thus eliminating any boundaries between languages. If a variable—in the course of further execution—is changed to reference a foreign object that has a different type or comes from a different language, the execution falls back to the AST interpreter and *generic access* resolves a new foreign object access.

RPython. Cross-language interoperability on top of RPython [3–5] allows the programmer to toggle between syntax and semantics of languages on the statement level. Barrett et al. describe a combination of Python and Prolog called Unipycation [3] or a combination of Python and PHP called PyHyp [5]. Unipycation and PyHyp compose languages by combining their interpreters. Both approaches glue the interpreters together on the language implementation level. The new interpreter is then compiled using a meta-tracing JIT. To share data across languages, Unipycation and PyHyp wrap objects using *adapters*. Like TruffleVM, the approach of Barrett et al. avoids compilation boundaries between languages and multi-language applications show good performance. In contrast to Barret et al.'s approach, however, TruffleVM is not restricted to a fixed set of languages. Unipycation and PyHyp both compose a specific pair of language implementations whereas *generic access* is a general mechanism for composing arbitrary Truffle language implementations. Unipycation, and PyHyp propose a more fine-grained language composition compared to

our approach. However, when languages are mixed at source code level, editors, compilers, and debuggers have to be adapted as well.

7.4 Multi-Language Semantics

The semantics of language composition is a well-researched area [1, 19, 20, 41, 54, 59]. However, most of these approaches do not have an efficient implementation. Our work partially bases on ideas from existing approaches (i.e., *like types* from Wrigstad et al. [59], Section 4.1.2) and, therefore, stands to complement such efforts.

8 CONCLUSIONS

In this article, we presented TruffleVM, a runtime with a set of language implementations that can efficiently execute multi-language applications. The language implementations of TruffleVM translate source code into a self-optimizing AST, which is interpreted and finally dynamically compiled to efficient machine code. TruffleVM can host managed high-level languages (JavaScript and Ruby) as well as unmanaged low-level languages (C).

A uniform approach of language implementation on the same VM is an essential factor for efficient cross-language interoperability. We compose different language implementations on the AST level via a language-agnostic mechanism, which we call *generic access*. Language implementations use language-independent messages to access foreign objects that are resolved at their first execution and transformed to efficient foreign-language-specific operations. *Generic access* is independent of languages, which allows adding new languages to TruffleVM without affecting existing languages. This approach leads to excellent performance of multi-language applications because of two reasons. First, message resolution replaces language-agnostic messages with efficient foreign-language-specific operations. Accessing foreign objects becomes as efficient as accessing objects of the host language. Second, the dynamic compiler can perform optimizations across language borders because these borders were removed by message resolution. We show that using heavyweight foreign data has a negative impact on performance, whereas using lightweight foreign data has a positive effect on performance. Our evaluation shows that the dynamic compiler of TruffleVM can eliminate many of the language and type checks that come with *generic access*.

We presented two case studies that evaluate *generic access*. First, we discussed seamless cross-language interoperability between JavaScript, Ruby, and C. TruffleVM allows programmers to directly access foreign objects using the operators of the host language. *Generic access* makes the mapping of access operations to messages largely the task of the language implementer rather than the task of the end programmer. Second, we used *generic access* to implement the C extensions API for TruffleRuby. TruffleC substitutes invocations of C extensions API functions and uses *generic access* for accessing Ruby objects instead. Our system is therefore compatible with MRI's C extensions API and can execute real-world applications. TruffleVM can be the basis for a wide variety of different areas of future research. Topics are, for example, multi-language concurrency and parallelism, cross-language inheritance, or cross-language debuggers.

ACKNOWLEDGMENTS

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for their valuable feedback on this work and on this article. We thank Daniele Bonetta, Stefan Marr, and Christian Wirth for feedback on this article. We especially thank Stephen Kell for significant contributions to our literature survey.

REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. 1989. Dynamic typing in a statically-typed language. In *Proceedings of the 16th Symposium on Principles of Programming Languages (POPL'89)*. ACM, New York, NY, 213–227. DOI: http://dx.doi.org/10.1145/75277.75296

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 93, 2 (2005), 449–466. DOI: http://dx.doi.org/10.1109/JPROC.2004.840305

[3] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. 2013. Unipycation: A case study in cross-language tracing. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL'13)*. ACM, New York, NY, 31–40. DOI: http://dx.doi.org/10.1145/2542142.2542146

[4] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. 2014. Approaches to interpreter composition. *CoRR* abs/1409.0757. Retrieved from http://arxiv.org/abs/1409.0757.

[5] Edd Barrett, Lukas Diekmann, and Laurence Tratt. 2015. Fine-grained language composition. *CoRR* abs/1503.08623. Retrieved from http://arxiv.org/abs/1503.08623.

[6] David M. Beazley and others. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*. 129–139.

[7] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1990. Lightweight remote procedure call. *ACM Trans. Comput. Syst.* 8, 1 (Feb. 1990), 37–55. DOI: http://dx.doi.org/10.1145/77648.77650

[8] Matthias Blume. 2001. No-longer-foreign: Teaching an ML compiler to speak C natively. *Electronic Notes in Theoretical Computer Science* 59, 1 (2001), 36–52.

[9] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09)*. ACM, New York, NY, 18–25. DOI: http://dx.doi.org/10.1145/1565824.1565827

[10] D. Box and C. Sells. 2002. Essential .NET. The Common Language Runtime, vol. I. (2002).

[11] David Chisnall. 2013. The challenge of cross-language interoperability. *Commun. ACM* 56, 12 (2013), 50–56. DOI: http://dx.doi.org/10.1145/2534706.2534719

[12] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and thread-safe objects for dynamically-typed languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, 642–659. DOI: http://dx.doi.org/10.1145/2983990.2984001

[13] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without regret: Reducing deoptimization meta-data in the graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'14)*. ACM, New York, NY, 187–193. DOI: http://dx.doi.org/10.1145/2647508.2647521

[14] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. 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)*. ACM, New York, NY, 1–10. DOI: http://dx.doi.org/10.1145/2542142.2542143

[15] ECMA-International. 2012. Standard ECMA-335. Common Language Infrastructure (CLI). Retrieved from http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[16] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. 1999. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*. ACM, New York, NY, 114–125. DOI: http://dx.doi.org/10.1145/317636.317790

[17] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM* 29, 3 (March 1986), 218–221. DOI: http://dx.doi.org/10.1145/5666.5673

[18] Yoshihiko Futamura. 1999. Partial evaluation of computation process—An approach to a compiler-compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391.

[19] Kathryn Gray, Robert Bruce Findler, and Matthew Flatt. 2005. Fine-grained interoperability through mirrors and contracts. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 231–245. DOI: http://dx.doi.org/10.1145/1094811.1094830

[20] Kathryn E. Gray. 2008. Safe cross-language inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'08)*. Lecture Notes in Computer Science, Vol. 5142. Springer Berlin, 52–75. DOI: http://dx.doi.org/10.1007/978-3-540-70592-5_4

[21] Matthias Grimmer, Stefan Marr, Mario Kahlhofer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Applying optimizations for dynamically-typed languages to java. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017)*. ACM, New York, NY, 12–22. DOI: http://dx.doi.org/10.1145/3132190.3132202

[22] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. 2014. TruffleC: Dynamic execution of C on a java virtual machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'14)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2647508.2647528

[23] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. 2013. An efficient native function interface for java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ'13)*. ACM, New York, NY, 35–44. DOI: http://dx.doi.org/10.1145/2500828.2500832

[24] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-safe execution of C on a java VM. In *Proceedings of the 10th Workshop on Programming Languages and Analysis for Security (PLAS'15)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2786558.2786565

[25] Matthias Grimmer, Chris Seaton, Roland Schatz, Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS'15)*. ACM, New York, NY.

[26] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically composing languages in a modular way: Supporting C extensions for dynamic languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, 1–13. DOI: http://dx.doi.org/10.1145/2724525.2728790

[27] Matthias Grimmer, Thomas Würthinger, Andreas Wöß, and Hanspeter Mössenböck. 2014. An efficient approach for accessing C data structures from javascript. In *Proceedings of 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE - Workshop on Programming Language Evolution, 2014 (ICOOOLPS'14)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/2633301.2633302

[28] Jeff Hardy. 2008. The dynamic language runtime and the iron languages. In *The Architecture of Open Source Applications, Volume II*, Amy Brown and Greg Wilson (Eds.). Retrieved from http://aosabook.org/en/index.html.

[29] Martin Hirzel and Robert Grimm. 2007. Jeannie: Granting java native interface developers their wishes. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA'07)*. ACM, New York, NY, 19–38. DOI: http://dx.doi.org/10.1145/1297027.1297030

[30] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, Pierre America (Ed.). Lecture Notes in Computer Science, Vol. 512. Springer Berlin, 21–38. DOI: http://dx.doi.org/10.1007/BFb0057013

[31] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI'92)*. ACM, New York, NY, 32–43. DOI: http://dx.doi.org/10.1145/143095.143114

[32] International Organization for Standardization. 2007. C99 Standard: ISO/IEX 9899:TC3. Retrieved from www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[33] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. 1997. GreenCard: A foreign-language interface for Haskell. In *Proc. Haskell Workshop*.

[34] Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM)*.

[35] Stephen Kell and Conrad Irwin. 2011. Virtual machines should be invisible. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH'11 Workshops)*. ACM, New York, NY, 289–296. DOI: http://dx.doi.org/10.1145/2095050.2095099

[36] F. Klock II. 2007. The layers of Larceny's foreign function interface. In *Scheme and Functional Programming Workshop*. Citeseer.

[37] Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. 2009. Debug all your code: Portable mixed-environment debugging. *SIGPLAN Not.* 44, 10 (Oct. 2009), 207–226. DOI: http://dx.doi.org/10.1145/1639949.1640105

[38] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[39] Sheng Liang. 1999. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional.

[40] Marr and Ducasse. 2015. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages; Applications (OOPSLA'15)*. ACM, New York, NY.

[41] Jacob Matthews and Robert Bruce Findler. 2007. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. ACM, New York, NY, 3–10. DOI: http://dx.doi.org/10.1145/1190216.1190220

[42] Erik Meijer and John Gough. 2001. Technical overview of the common language runtime. *Language* 29 (2001), 7.

[43] Mozilla Developer Network. 2014. XPCOM Specification. Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/XPCOM.

[44] Object Management Group. 2014. Common Object Request Brooker Architecture (CORBA) Specification. Retrievced from http://www.omg.org/spec/CORBA/3.3/.

[45] John Reppy and Chunyan Song. 2006. Application-specific foreign-interface generation. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*. ACM, New York, NY, 49–58. DOI: http://dx.doi.org/10.1145/1173706.1173714

[46] John R. Rose and Hans Muller. 1992. Integrating the scheme and C languages. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP'92)*. ACM, New York, NY, 247–259. DOI: http://dx.doi.org/10.1145/141471.141559

[47] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*. ACM, 1–13.

[48] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5 (2007).

[49] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the 6th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL'12)*. ACM, New York, NY, 49–58. DOI: http://dx.doi.org/10.1145/2414740.2414750

[50] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala (SCALA'13)*. ACM, New York, NY, Article 9, 8 pages. DOI: http://dx.doi.org/10.1145/2489837.2489846

[51] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY, Article 165, 10 pages. DOI: http://dx.doi.org/10.1145/2544137.2544157

[52] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. 2005. Inlining java native calls at runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE'05)*. ACM, New York, NY, 121–131. DOI: http://dx.doi.org/10.1145/1064979.1064997

[53] TC39. 2016. Official ECMAScript Conformance Test Suite. Retrieved from https://github.com/tc39/test262.

[54] Valery Trifonov and Zhong Shao. 1999. *Safe and Principled Language Interoperation*. Springer.

[55] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. 2001. Overview of the CORBA component model. In *Component-Based Software Engineering*. Addison-Wesley Longman, 557–571.

[56] Michal Wegiel and Chandra Krintz. 2010. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, New York, NY, 223–240. DOI: http://dx.doi.org/10.1145/1869459.1869479

[57] Christian Wimmer and Stefan Brunthaler. 2013. ZipPy on truffle: A fast and simple implementation of python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, Applications: Software for Humanity (SPLASH'13)*. ACM, New York, NY, 17–18. DOI: http://dx.doi.org/10.1145/2508075.2514572

[58] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. 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)*. ACM, New York, NY, 133–144. DOI: http://dx.doi.org/10.1145/2647508.2647517

[59] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM, New York, NY, 377–388. DOI: http://dx.doi.org/10.1145/1706299.1706343

[60] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. 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)*. ACM, New York, NY, 662–676. DOI: http://dx.doi.org/10.1145/3062341.3062381

[61] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, 187–204. DOI: http://dx.doi.org/10.1145/2509578.2509581

[62] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS'12)*. ACM, New York, NY, 73–82. DOI: http://dx.doi.org/10.1145/2384577.2384587

[63] Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. 2014. Accelerating iterators in optimizing AST interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages; Applications (OOPSLA'14)*. ACM, New York, NY, 727–743. DOI: http://dx.doi.org/10.1145/2660193.2660223

Received March 2016; revised October 2017; accepted February 2018