High-Performance Cross-Language Interoperability in a Multi-language Runtime

Matthias Grimmer
Johannes Kepler University Linz, Austria
matthias.grimmer@jku.at

Chris Seaton
Oracle Labs, United Kingdom
chris.seaton@oracle.com

Roland Schatz
Oracle Labs, Austria
roland.schatz@oracle.com

Thomas Würthinger
Oracle Labs, Switzerland
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Programmers combine different programming languages because it 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, existing cross-language mechanisms suffer from complex interfaces, insufficient flexibility, or poor performance.

We present the TruffleVM, a multi-language runtime that allows composing different language implementations in a seamless way. It reduces the amount of required boilerplate code to a minimum by allowing programmers to access foreign functions or objects by using the notation of the host language. We compose language implementations that translate source code to an intermediate representation (IR), which is executed on top of a shared runtime system. Language implementations use language-independent messages that the runtime resolves at their first execution by transforming them to efficient foreign-language-specific operations. The TruffleVM avoids conversion or marshaling of foreign objects at the language boundary and allows the dynamic compiler to perform its optimizations across language boundaries, which guarantees high performance. This paper presents an implementation of our ideas based on the Truffle system and its guest language implementations JavaScript, Ruby, and C.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DLS '15, October 25-30, 2015, Pittsburgh, PA, USA.
Copyright © 2015 ACM 978-1-4503-3690-1/15/10...$15.00.
http://dx.doi.org/10.1145/2816707.2816714

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Code generation, Interpreters, Compilers, Optimization

Keywords cross-language; language interoperability; virtual machine; optimization; language implementation

1. Introduction

The likelihood that a program is entirely written in a single language is lower than ever [8]. Composition of languages is important for mainly three reasons: programmers can use the *most suitable language for a given problem*, can *gradually migrate* existing projects from one language to another, and can *reuse* existing source code.

There exists no programming language that is best for all kinds of problems [2, 8]. High-level languages allow representing a subset of algorithms efficiently but sacrifice low-level features such as pointer arithmetic and raw memory accesses. A typical example is business logic written in a high-level language such as JavaScript that uses a database driver written in a low-level language such as C. Cross-language interoperability allows programmers to pick the most suitable language for a given part of a problem. Existing approaches cater primarily to composing two specific languages, rather than arbitrary languages. These pairwise efforts restrict the flexibility of programmers because they have to select languages based on given cross-language interfaces.

Cross-language interoperability reduces the risks when migrating software from one language to another. For example, programmers can gradually port existing C code to Ruby, rather than having to rewrite the whole project at once. However, existing approaches (e.g. Ruby's C extension mechanism) require the programmer to write wrapper code to integrate foreign code in a project, which adds a

maintenance burden and also distracts the programmer from the actual task at hand.

Finally, cross-language interoperability allows reusing existing source code. Due to the large body of existing code it is not feasible to rewrite existing libraries in a different language. A more realistic approach is to use a cross-language interoperability mechanism that allows reusing this existing code. However, existing solutions convert data at the language border [34], use generic data representations that cannot be optimized for an individual language [35], or cannot widen the compilation scope across languages [24], which introduces a runtime overhead. Programmers have to sacrifice performance if components, written in different languages, are tightly coupled.

In this paper we present the TruffleVM, a multi-language runtime that composes individual language implementations that run on top of the same virtual machine, and share the same style of IR. Multi-language applications can access foreign objects and can call foreign functions by simply using the operators of the host language, which makes writing multi-language applications easy. In our system, a multi-language program just uses different files for different programming languages. The TruffleVM makes language boundaries mostly invisible because the runtime implicitly bridges them. Hence, we can reduce the amount of required boiler-plate code to a minimum. For example, the JavaScript statement in Figure 1 can access the C structure `obj` as if it were a regular JavaScript object. Only if semantics of languages fundamentally differ, programmers should need to revert to an API and therefore an explicit foreign object access.

We are convinced that a well-designed cross-language interoperability mechanism should not only target a fixed set of languages but should be general enough to support interoperability between arbitrary languages. The TruffleVM uses the *dynamic access*, which is an interoperability-mechanism that allows combining arbitrary languages by composing their implementations on top of the TruffleVM. The dynamic access is independent of programming languages and their implementations for the TruffleVM. It is possible to add new languages to the TruffleVM without affecting the existing language implementations.

Also, crossing language boundaries does not introduce runtime overhead. First, we do not require a common representation of data for all languages (e.g., in contrast to the CLR [6]), but each language can use the data structures that meet the requirements of this individual language best. For example, a C implementation can allocate raw memory on the native heap while a JavaScript implementation can use dynamic objects on a managed heap. Second, each language can access foreign objects or invoke foreign functions without introducing run-time overhead. The dynamic access enables dynamic compilers to optimize a foreign object access like any regular object access and also to widen the compilation scope across language boundaries and thus to perform cross-language optimizations. For example, we can inline a JavaScript function into a caller that is written in C.

JavaScript Code:

```
var a = obj.value;
```

C Code:

```
struct S {
  int value;
};
struct S * obj;
```

Figure 1: JavaScript can access C data structures.

As a case study, we compose the languages JavaScript, Ruby, and C. This allows us to discuss the following core aspects:

1. The host language implementation maps language-specific operations to messages, which are used to access objects in a language-independent way. For example, the JavaScript implementation maps the property access of Figure 1 to a message.
2. The foreign language maps these messages to foreign-language specific accesses. The TruffleVM uses this mapping to replace messages by efficient foreign-language-specific operations at their first execution. For example, the TruffleVM replaces the message to read the member `value` by a structure access.
3. Finally, we discuss how we can bridge differences between JavaScript, Ruby, and C. These differences are: object-oriented vs. non-object-oriented; dynamically typed vs. statically typed; explicit memory management vs. automatic memory management; safe memory access vs. unsafe memory access.

We present a performance evaluation using multi-language benchmarks. We simulate using a new language in an existing code base, or updating a legacy code base gradually by translating parts of well-established benchmarks from language A to language B. We use the languages JavaScript, Ruby, and C and can show that language boundaries do not cause a performance overhead. In summary, this paper contributes the following:

- We describe a multi-language runtime that composes different language implementations. Rather than composing a pair of languages (e.g., in contrast to a foreign function interface (FFI)) we support interoperability between arbitrary languages. We show how language implementations map language-dependent operations to language-independent messages and vice versa.
- We list the different semantics of JavaScript, Ruby, and C and explain how we bridge these differences.
- We show the simplicity and extensibility of the TruffleVM. Also, we evaluate the performance of the TruffleVM using non-trivial multi-language benchmarks.

2. System Overview

The TruffleVM targets language implementations, hosted by a general-purpose VM. In the context of this paper, a *language implementation* (LI) translates the source code into an IR. We could identify the following requirements:

Compatible IR: LIs have to translate the applications' source code to compatible (but not necessarily equal) IRs.

Rewriting capability: LIs need a rewriting capability that allows them to replace IR snippets with different IR snippets at run time.

Sharable data: The data structures used by LIs to represent the data of an application need to be accessible by all LIs.

Dynamic compilation: The dynamic compiler of the host VM compiles the IR of a program to highly efficient machine code at runtime.

These requirements exist for bytecode-based VMs (e.g. a JVM): They use a common IR, which is bytecode. Techniques such as bytecode quickening [7] provide the rewriting capability and all languages share a common heap for their data.

The same requirements also exist for Truffle [39], a platform for implementing high-performance LIs in Java. We use Truffle for our case study because there are many LIs available; including JavaScript, Ruby, and C. Truffle LIs are abstract syntax tree (AST) interpreters, running on top of a Java Virtual Machine. Source code is compiled to an AST, which is then dynamically executed by the Truffle framework. Every node has a method that evaluates the node. By calling these methods recursively, the whole AST is evaluated. All nodes extend a common base class `Node`.

Truffle ASTs are self-optimizing in the sense that AST nodes can speculatively rewrite themselves with *specialized* variants [38] at run time, e.g., based on profile information obtained during execution such as type information. If these speculative assumptions turn out to be wrong, the specialized tree can be reverted to a more generic version that provides functionality for all possible cases. Truffle guest languages use self-optimization via tree rewriting as a general mechanism for dynamically optimizing code at run-time.

After an AST has become stable (i.e., no more rewritings occur) and when the execution frequency has exceeded a predefined threshold, Truffle dynamically compiles the AST to highly optimized machine code. The Truffle framework uses the Graal compiler [27] (which is part of the Graal VM) as its dynamic compiler. The compiler inlines node execution methods of a hot AST into a single method and performs aggressive optimizations over the whole tree. It also inserts *deoptimization points* [20] in the machine code where the speculative assumptions are checked. If they turn out to be wrong, control is transferred back from compiled code to the interpreted AST, where specialized nodes can be reverted to a more generic version. The Graal VM [27] is a minor modification of the HotSpot VM: it adds the Graal compiler, but reuses all other parts (including the garbage collector and the interpreter) from the HotSpot VM. Figure 2 shows the layers of a Truffle-based system.

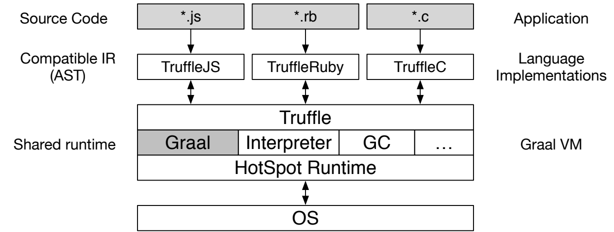


Figure 2: Layers of a Truffle-based system: TruffleJS, TruffleRuby, and TruffleC are hosted by the Truffle framework on top of the Graal VM.

In this paper we use three LIs on top of Truffle:

TruffleJS is a state-of-the-art JavaScript engine that is fully compatible with the JavaScript standard. TruffleJS' speedup varies between $0.3\times$ and $3\times$ (average: $0.66\times$) compared to Google's V8; between $0.4\times$ and $1.3\times$ (average: $0.65\times$) compared to Mozilla's Spidermonkey; between $0.9\times$ and $27\times$ (average: $5.8\times$) compared to Nashorn as included in JDK 8u5 (results taken from [36]).

TruffleRuby is a Ruby implementation, which is an experimental option of JRuby. TruffleRuby performs well compared to existing Ruby implementations. Its speedup varies between $2.7\times$ and $38.2\times$ (average: $12.7\times$) compared to MRI; between $1.3\times$ and $39.8\times$ (average: $6.2\times$) compared to Rubinius; between $2.7\times$ and $17\times$ (average: $6.6\times$) compared to JRuby; and between $1\times$ and $7.6\times$ (average: $3.5\times$) compared to Topaz (results taken from [36]).

TruffleC is a C implementation on top of Truffle [14, 16] and can dynamically execute C code. TruffleC's speedup varies between $0.6\times$ and $1.1\times$ (average: $0.81\times$) compared to the best performance of the GNU C Compiler (results taken from [16]). TruffleC does not yet support the full C standard. However, there are no conceptual limitations and future work will address completeness issues.

Foreign Object Access: In the context of this paper, an *object* is a non-primitive entity of a user program, which we want to share across different LIs. Examples include data (such as JavaScript objects, Ruby objects, or C pointers), as well as functions, classes or code-blocks. If the Ruby implementation accesses a Ruby object, the object is considered a *regular object*. If Ruby (*host language*, LI_{Host}) accesses a C structure, the C structure is considered a *foreign object* (we call C the *foreign language*, $\text{LI}_{\text{Foreign}}$). *Object accesses*

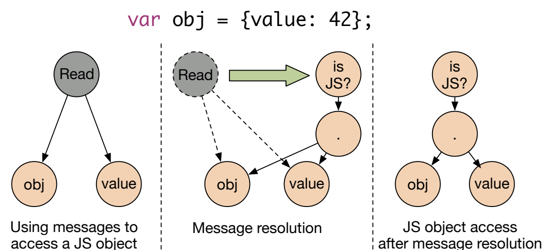


Figure 3: The dynamic access accesses a JavaScript object using messages; Message resolution replaces the *Read* message by a direct access.

are operations that an LI_{Host} can perform on objects, e.g., method calls, property accesses, or field reads. We base our work on the *dynamic access*, which is a message-based approach to access foreign objects [13, 17].

Truffle LIs 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. Hence, each LI uses language-specific nodes to access *regular objects*. To access a *foreign object*, an LI_{Host} can use the dynamic access: The dynamic access defines a set of language-independent *messages*, used to access foreign objects. The left part of Figure 3 shows a Truffle AST snippet that reads the `value` property of a JavaScript object `obj` using a *Read* message.

Message resolution: The LI_{Host} uses a message to access a foreign object, which has exactly one language it belongs to. The TruffleVM uses this foreign language $\text{LI}_{\text{Foreign}}$ to resolve the message to a foreign-language-specific AST snippet (*message resolution*) upon first execution. The $\text{LI}_{\text{Foreign}}$ provides an AST snippet that can be inserted into the host AST as a replacement for the messages. This snippet contains foreign-language-specific operations that directly access the receiver. Thus, message resolution replaces language-independent messages by language-specific operations. 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 3, message resolution inserts a *JSReadProperty* node (a JavaScript-specific AST node that reads the property of a JavaScript object; in Figure 3 we use the abbreviated label `"."` for this node). Before the AST accesses `obj`, it checks if `obj` is a JavaScript object (*is JS?*). 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, the TruffleVM embeds the different language-specific AST snippets in a chain like an inline cache [19] and therefore avoids a loss in performance.

Primitive types: Besides objects, also values with a primitive type can be shared across languages. The work of [17] defines a set of *shared primitive types*. Truffle languages map language-specific primitive types from and to this set, which allows exchanging primitive values.

In [17], we describe how this approach is used to compose the C and Ruby implementations in order to support C extensions for Ruby. The runtime that we present in this paper composes arbitrary languages rather than a *pair of languages* (Ruby and C). Section 6 discusses in detail how our approach differs from traditional FFIs on top of Truffle.

3. Multi-Language Composition

The 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 parsers of each LI without modification. Syntactical and grammatical combination is out of scope for this work.

Programmers can export data and functions to a multi-language scope and also import data and functions from this scope. This allows programmers to explicitly share data among other languages. JavaScript, Ruby, and C provide built-ins that allow exporting and importing data to and from the multi-language scope.

3.1 Implicit Foreign Object Accesses

The TruffleVM allows programmers to access foreign objects transparently. The VM maps *host-language-specific operations* to *language-independent messages*, which are then mapped back to *foreign-language-specific operations*. Truffle LIs compile source code to a tree of nodes, i.e., an AST. N^A and N^B define finite sets of nodes of LIs A and B. Each node has $r : N^A \to \mathbb{N}$ 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. An AST $t \in T_{N^A}$ is a tree of nodes $n \in N^A$. By $n(t_1, ..., t_k)$ we denote a tree with root node $n \in N^A$ and k sub-trees $t_1, \dots, t_k \in T_{N^A}$, where $k = r(n)$.

The dynamic access defines a set of messages, which are modeled as language-independent nodes N^{Msg}:

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

If the LI_{Host} A uses messages to access a foreign object, the tree $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ consists of language-specific nodes N^A and language-independent nodes N^{Msg}.

To compose JavaScript, Ruby, and C we use the messages $n \in N^{\text{Msg}}$ where the sub-trees $t_1, \dots, t_k \in T_{N^A \cup N^{\text{Msg}}}$ of $n(t_1, ..., t_k)$ evaluate to the arguments of the message:

Read: Truffle LIs use the *Read* message to access a field of an object or an element of an array. It can also be used

to access methods of classes or objects, i.e., to lookup executable methods from classes and objects.

$$\text{Read}(t_{\text{rec}}, t_{\text{id}}) \in T_{N^A \cup N^{\text{Msg}}} \quad (2)$$

The first subtree t_{rec} evaluates to the receiver of the *Read* message, the second subtree t_{id} to a name or an index.

Write: An LI uses the *Write* message to set the field of an object or the element of an array. It can also be used to add or change the methods of classes and objects.

$$\text{Write}(t_{\text{rec}}, t_{\text{id}}, t_{\text{val}}) \in T_{N^A \cup N^{\text{Msg}}} \quad (3)$$

The first subtree t_{rec} evaluates to the receiver of the *Write* message, the second subtree t_{id} to a name or an index, and the third subtree t_{val} to a value.

Execute: LIs execute methods or functions using an *Execute* message.

$$\text{Execute}(t_{\text{f}}, t_1, \ldots, t_i) \in T_{N^A \cup N^{\text{Msg}}} \quad (4)$$

The first subtree t_{f} evaluates to the function/method itself, the other arguments $t_1, \ldots, t_i$ to 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 a wrapper object and produces a primitive value. LIs use this message to unbox a boxed value whenever a primitive value is required.

$$\text{Unbox}(t_{\text{rec}}) \in T_{N^A \cup N^{\text{Msg}}} \quad (5)$$

The subtree t_{rec} evaluates to the receiver object.

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

$$\text{IsNull}(t_{\text{rec}}) \in T_{N^A \cup N^{\text{Msg}}} \quad (6)$$

The subtree t_{rec} evaluates to the receiver object.

3.1.1 Mapping language-specific operations to messages

If language A encounters a foreign object at runtime and the regular object access operations cannot be used, then language A uses the dynamic access. The LI_{A} maps an AST with a language-specific object access $t_a \in T_{N^A}$ to an AST with a language-independent access $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ using a the function f_A:

$$T_{N^A} \xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \quad (7)$$

The function f_A replaces the language specific access and inserts a language independent access instead. The other parts of the AST t_a remain unchanged. An LI_{Host} that accesses foreign objects has to define this function f_A.

Consider the example in Figure 4 (showing a property access in JavaScript: `obj.value`), f_{JS} replaces the JavaScript-specific object access (*JSReadProperty*, node `"."` in the AST) with a language-independent *Read* message (see left part of Figure 4).

$$\text{JSReadProperty}(t_{\text{obj}}, t_{\text{value}}) \xmapsto{f_{\text{JS}}} \text{Read}(t_{\text{obj}}, t_{\text{value}}) \quad (8)$$

Rather than using a *JSReadProperty* node to access the `value` property of the receiver `obj`, the JavaScript implementation uses a *Read* message.

3.1.2 Mapping messages to language-specific operations

The TruffleVM uses the $\text{LI}_{\text{Foreign}}$ B to map the host AST with a language-independent access $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ to an AST with a foreign language-specific access $t_{a,b} \in T_{N^A \cup N^B}$ using the function g_B:

$$T_{N^A \cup N^{\text{Msg}}} \xrightarrow{g_B} T_{N^A \cup N^B} \quad (9)$$

Message resolution removes the language-independent access and inserts a language specific access instead, which produces an AST that consists of nodes $N^A \cup N^B$. The other parts of $t_{a,m}$ remain unchanged. With respect to the example in Figure 4, the TruffleVM replaces the *Read* message with a C-specific access operation upon its first execution. More specifically, it uses g_C to replace the *Read* message with a *CMemberRead* node (node `"->"` in the AST):

$$\text{Read}(t_{\text{obj}}, t_{\text{value}}) \xmapsto{g_C} \text{CMemberRead}(t_{\text{obj}}, t_{\text{value}}) \quad (10)$$

The result is a JavaScript AST that embeds a C access operation $t_{\text{JS,C}} \in T_{N^{\text{JS}} \cup N^C}$.

Language implementers have to define the functions f_A and g_B; the TruffleVM then creates the pairwise combination automatically by composing these functions at runtime:

$$g_B \circ f_A : T_{N^A} \to T_{N^A \cup N^B} \quad (11)$$

When accessing foreign objects, the TruffleVM automatically creates an AST $t_{a,b} \in T_{N^A \cup N^B}$ where the main part is specific to language A and the foreign object access is specific to language B.

For further reference, we provide a detailed table that lists all mappings from language-specific operations to messages and vice versa[1].

3.1.3 Limitation

Consider a function f_A that maps an AST with a language-specific object access $t_a \in T_{N^A}$ to an AST with a language-independent access $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ and a function g_B that

[1] `http://ssw.jku.at/General/Staff/Grimmer/TruffleVM_table.pdf`

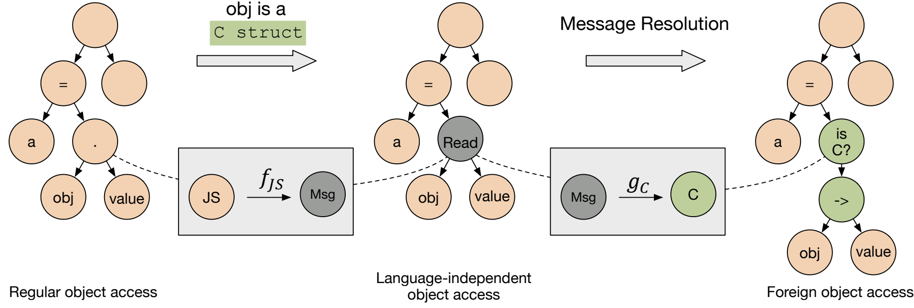


Figure 4: Accessing a C structure form JavaScript; Message resolution inserts a C struct access into a JavaScript AST.

maps $t_{a,m}$ to an AST with a foreign-language-specific object access $t_{a,b} \in T_{N^A \cup N^B}$:

$$
\begin{aligned}
T_{N^A} &\xrightarrow{f_A} T_{N^A \cup N^{\text{Msg}}} \\
T_{N^A \cup N^{\text{Msg}}} &\xrightarrow{g_B} T_{N^A \cup N^B}
\end{aligned}
\tag{12}
$$

When composing f_A and g_B three different cases can occur:

1. If g_B is defined for $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ a foreign object can be accessed implicitly. The TruffleVM can replace the language-independent object access with a B-specific access.
2. If g_B is *not* defined for $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$, we report a runtime error with a high-level diagnostic message. The foreign object access is not supported by the foreign language. For example, if JavaScript accesses the length property of a C array, we report an error. C cannot provide length information for arrays.
3. A foreign object access might not be expressible in A, i.e., one wants to create $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$ 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.

3.2 Explicit Foreign Object Accesses

A host language might not provide syntax for a specific foreign object access. Consider the JavaScript array `arr` of Figure 5, which is used in a C program: C does not provide syntax for accessing the length property of an array.

We overcome these issues by exposing the dynamic access to the programmer. Using this interface, the programmer can fall back to an *explicit* foreign object access. The programmer sends a message directly in order to access a foreign object. In other words, this interface allows programmers to handcraft the foreign object access of $t_{a,m} \in T_{N^A \cup N^{\text{Msg}}}$.

Every LI on top of Truffle has an API for explicit message sending. For example, to access the `length` property of a JavaScript array from C (see Figure 5), the programmer

C Code:

```
int arr = // …
int length = Read(arr, "length");
```

JavaScript Code:

```
var arr = new Array(5);
```

Figure 5: C accessing the length property of a JavaScript array.

uses the built-in C function `Read`. The C implementation substitutes this `Read` invocation by a *Read* message.

4. Different Language Paradigms and Features

In this section we describe how we map the different paradigms and features of the languages JavaScript, Ruby, and C to messages. It is not possible to cover all features and paradigms of the vast amount of languages that are available. Hence, we focus on JavaScript, Ruby, and C and show how we map the concepts of these fundamentally different languages to the dynamic access, which demonstrates the feasibility of the TruffleVM. We explain how we deal with dynamic typing versus static typing, object-oriented versus non-object oriented semantics, explicit versus automatic memory management, as well as safe versus unsafe memory accesses.

4.1 Dynamic Typing vs. Static Typing

In [37], Wrigstad et al. describe an approach called *like types* — they show how dynamic objects can be used in a statically typed language by binding them to *like-type* variables. Occurrences of *like-type* variables are checked statically, but their usage is checked dynamically. The TruffleVM is similar, except that in our case any pointer variable in C can be bound to a foreign object:

We bind foreign dynamic objects to pointer variables that are associated with static type information. If a pointer is bound to a dynamically typed value, we check the usage dynamically, i.e., upon each access we check whether the operation on the foreign object is possible. We report a runtime error otherwise.

Figure 6 shows a C program, which uses a JavaScript object `jsObject`. The C code associates `jsObject` with the static type `struct JsObject*`, which is defined by the programmer (Figure 6, Label 1). When the C code accesses the JavaScript object (Label 3), we check whether `bar` exists and report an error otherwise.

4.2 Object-Oriented vs. Non-Object Oriented Semantics

The object-oriented programming paradigm allows programmers to create objects that contain both data and code, known as *fields* and *methods*. Also, objects can extend each other (e.g. class-based inheritance or prototype-based inheritance) — when accessing fields or methods, the object does a lookup and provides a field value or a method.

The TruffleVM uses the dynamic access, which retains this mechanism. Consider the method invocation (from C to JavaScript) at Label 3 in Figure 6: TruffleC maps this access to the following messages:

$$\begin{aligned} &\text{CCall}(\text{CMemberRead}(t_{\text{jsObject}}, t_{\text{bar}}), t_{\text{jsObject}}, t_{84}) \\ &\xmapsto{f_C} \text{Execute}(\text{Read}(t_{\text{jsObject}}, t_{\text{bar}}), t_{\text{jsObject}}, t_{84}) \end{aligned} \tag{13}$$

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

$$\begin{aligned} &\text{Execute}(\text{Read}(t_{\text{jsObject}}, t_{\text{bar}}), t_{\text{jsObject}}, t_{84}) \\ &\xmapsto{g_{JS}} \text{JSCall}(\text{JSReadProperty}(t_{\text{jsObject}}, t_{\text{bar}}), t_{\text{jsObject}}, t_{84}) \end{aligned} \tag{14}$$

A method call in an object-oriented language passes the *this* object as an implicit argument. Non-object oriented languages that invoke methods therefore need to explicitly pass the *this* object. For example, the JavaScript function `bar` (see Figure 6) expects the *this* object as the first argument. Hence, the first argument of the method call in C (Label 3) is the *this* object `jsObject`.

Vice versa, the signature of a non-object-oriented function needs to contain the *this* argument if the caller is an object-oriented language. For example, if JavaScript calls the C function `foo` (Label 4), JavaScript passes the *this* object as the first argument. The signature of the C function `foo` (Label 5) explicitly contains the *this* object. A wrong number of arguments causes a runtime error.

Future work: The TruffleVM currently does not support cross-language inheritance, i.e., class-based inheritance or prototype-based inheritance is only possible with objects that originate from the same language. We are convinced that the TruffleVM is extensible in this respect and therefore our future research will focus on inheritance across language boundaries.

4.3 Explicit vs. Automatic Memory Management

Truffle LIs are running on a shared runtime and can exchange data, independent of whether it is managed or unmanaged:

Unmanaged allocations: Truffle LIs keep unmanaged allocations on the native heap, which is not garbage collected. For example, TruffleC allocates data on the native heap. TruffleC represents all pointers (pointers to values, arrays, structures, and functions) as managed Java objects of type `CAddress` that wrap a 64-bit address value [15] and attach type information to the address value. TruffleC uses this type information to resolve messages and provide AST snippets that can access the data, stored on the native heap. When accessing a `CAddress` object via a dynamic access, the access will resolve to a raw memory accesses. The dynamic access allows accessing unmanaged data from a language that otherwise only uses managed data.

Managed allocations: The JavaScript and Ruby implementations allocate objects on the Java heap. If an application binds a managed object to a C variable, then 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.

Trade-offs: If a pointer variable of an unmanaged language 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, referencing a managed object in any other way (e.g. `jsArray = jsArray + 1`).

Second, pointer variables referencing managed objects cannot be cast 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. We report a high-level error-message in that case.

4.4 Safe vs. 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:

Unsafe accesses: If an unsafe language (such as C) shares data with a safe language (such as JavaScript), all access operations are *unsafe*. For example, accessing a C array in JavaScript is unsafe. If the index is out of bounds, the access has an undefined behavior (as defined by the C specification). However, accessing a C array directly is more efficient than accessing dynamic JavaScript array because less runtime checks are required.

Safe accesses: Accessing data structures of a safe language (such as JavaScript) from an unsafe language (such

```
① struct JsObject {                                  var jsObject = {  ②
     void (*bar)(void *receiver, int b);               bar: function (b) {
   };                                                    // …
   struct JsObject *jsObject = // …                    }
                                                     }
③ jsObject->bar(jsObject, 84);                       // …

⑤ void foo(void *receiver, int a);                   foo(42);          ④
```

Figure 6: Foreign object definition in C and an object-oriented object access.

as C) is *safe*. For example, accessing a JavaScript array in 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.

5. Evaluation

In this section we discuss why we claim that the TruffleVM improves the current state-of-the-art in cross-language interoperability. We focus on *simplicity* of foreign object accesses, as well as on *extensibility* of the TruffleVM with respect to new languages. We also evaluate the *performance* of a multi-language application.

5.1 Simplicity

Accessing foreign objects with the operators of the host language improves simplicity. Programmers are not forced to write boiler-plate code as long as an object access can be mapped from language A to language B ($t_a \xmapsto{f_A} t_{a,m} \xmapsto{g_B} t_{a,b}$). We make the mapping of language operations to messages largely the task of language implementers rather than the task of application programmers.

If not otherwise possible, programmers can also handcraft accesses to foreign objects. All languages expose an API that allows programmers to explicitly access foreign objects using the dynamic access.

We modified single-language benchmarks such that parts of them were written in a different language. The other parts did not have to be changed, because accesses to foreign-language objects can simply be written in the language of the host. The only extra code that we needed was for importing and exporting objects form and to the multi-language scope.

5.2 Extensibility

Many existing cross-language mechanisms cannot be extended to other languages. For example, FFIs are designed for a set of two languages and it is hard to extend them to include other languages. We can add new languages to the TruffleVM if they support the following:

Language-independent access: The LI_{Host} has to map an AST with language-specific accesses to an AST with language-independent accesses, i.e., an LI_{Host} has to define $T_{N^A} \xrightarrow{f_A} T_{N^A \cup N^{Msg}}$. If the semantics of a new language do not allow the mapping of certain operations to messages, language implementers can still provide an API to support an explicit foreign object access (see Section 3.2), which limits the implicit foreign object access but still guarantees good performance.

Resolve a language-independent access: An $\text{LI}_{\text{Foreign}}$ has to define a mapping from an AST with language-independent accesses to an AST with foreign language-specific accesses $T_{N^A \cup N^{Msg}} \xrightarrow{g_B} T_{N^A \cup N^B}$. This mapping allows the language implementer to decide how other languages can access objects.

Multi-language scope: The LI has to provide infrastructure for the application programmer to export and import objects to and from the multi-language scope.

Implementing these requirements for an existing Truffle language is little effort: A single programmer was able to implement the dynamic access for TruffleRuby within one week and we could add it to the TruffleVM.

5.3 High Performance

We evaluated the TruffleVM with a number of benchmarks that show how multi-language applications perform compared to single-language applications.

Benchmarks: For this evaluation we use benchmarks that heavily access objects and arrays. The benchmarks (the SciMark benchmarks[2] and benchmarks from the Computer Language Benchmarks Game[3]) 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 (`TS`), generate and write random DNA sequences (`Fasta`), and solve the towers of Hanoi problem (`Tower`).

Experimental Setup: The benchmarks were executed on an Intel Core i7-4770 quad-core 3.4GHz CPU running 64 Bit Debian 7 (Linux3.2.0-4-amd64) with 16 GB of memory. We base the TruffleVM on Graal revision `bf586af6fa0c` from the official OpenJDK Graal repository[4]. In this evaluation we show the score for each benchmark and its configu-

[2] `http://math.nist.gov/scimark2/index.html`

[3] `http://benchmarksgame.alioth.debian.org/`

[4] `http://openjdk.java.net/projects/graal/`

ration, which is the proportion of the execution count of the benchmark and the time needed ($executions/second$).

For this evaluation we are interested in peak performance of long running applications. Hence, we executed every benchmark 10 times with the same parameters after an initial warm-up of 50 iterations to arrive at a stable peak performance and calculated the averages for each configuration using the arithmetic mean.

Using foreign objects has no influence on compile time. We compile ASTs where messages are already resolved, i.e., for the compiler there is no difference between a foreign or a regular object access. Also, message resolution happens at the first execution and happens only once. Compared to the single-language implementations, the warm-up time did not change. A general evaluation of warm-up performance of Truffle LIs is out of scope for this work.

The error bars in the charts of this section show the standard deviation. The x-axis of the charts in Figure 7, 8, and 9 shows the different benchmarks. The y-axis of the charts in Figure 7, 8, and 9 shows the average scores (higher is better) of the benchmarks. Where we summarize across different benchmarks we report a geometric mean [10].

5.3.1 Results of single-language benchmarks

We compare the performance of the individual languages on our benchmarks. The results in Figure 7 are normalized to the C performance. This evaluation shows that JavaScript code is on average 37% slower and Ruby code on average 66% slower than C code. We explain the differences as follows:

C data accesses do not require runtime checks (such as array bounds checks), but the memory is accessed 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 and Ruby perform better than C on *Treesort*. 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, which explains why Ruby is in general slower than JavaScript or C.

5.3.2 Results of multi-language benchmarks

We modified the benchmarks by extracting all array and object allocations into factory functions. We then replaced these factory functions with implementations in different languages, making the benchmarks multi-language applications.

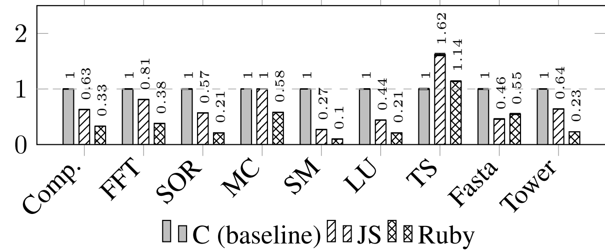
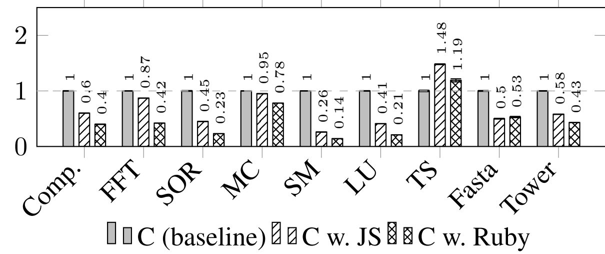
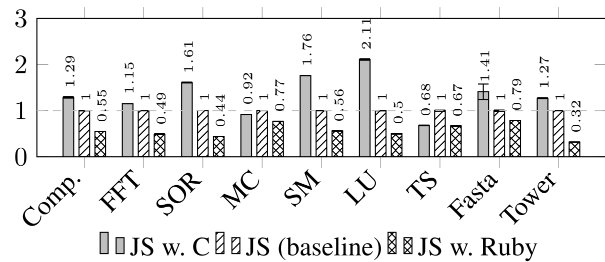


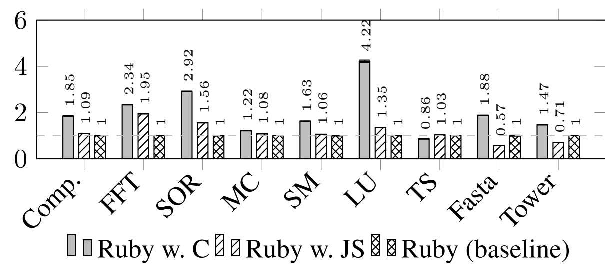
Figure 7: Performance of individual languages on our benchmarks (normalized to C performance; higher is better).



(a) Main part in C.



(b) Main part in JavaScript.



(c) Main part in Ruby.

Figure 8: Performance of multi-language applications (higher is better).

We grouped our evaluations (Figure 8) 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. We then replaced the factory functions by implementations in a different language and compared the multi-language configurations to the baseline of each group.

These multi-language applications heavily access foreign arrays/objects and call foreign functions, which makes them good candidates for our evaluation.

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 in JavaScript or Ruby applications improves the run-time performance. However, an allocation with `calloc` is more expensive than an allocation on the Java heap. The *Treesort* benchmark allocates objects in its main loop, hence, factory functions in JavaScript or Ruby perform better than factory functions written in C.

JS Objects: TruffleJS uses a dynamic object implementation where each access involves run-time 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 JavaScript objects perform worse than C objects.

Ruby Objects: TruffleRuby's dispatch mechanism for accessing objects introduces a performance overhead compared to JavaScript and C, which explains why Ruby objects are in general slower than JavaScript objects or C objects.

Our evaluation shows that the performance of a multi-language program depends on the performance of the individual language parts. Using heavy-weight foreign data has a negative impact on performance: Figure 8a and 8b show that using heavy-weight Ruby objects in C or JavaScript causes a slowdown of up to $7\times$. On average, using Ruby data reduces the C performance by a factor of 2.5 and the JavaScript performance by a factor of 1.8. On the other hand, using efficient foreign data has a positive effect on performance: For example, Figure 8b and 8c show that using efficient C data in JavaScript or Ruby can improve performance by up to a factor of 4.22. On average, using C data improves the JavaScript performance by a factor of 1.29 and the Ruby performance by a factor of 1.85.

5.3.3 Removing language boundaries

Message resolution as part of the dynamic access allows Truffle's dynamic compiler to apply its optimizations across language boundaries (*cross-language inlining*). Message resolution removes language boundaries by merging different language-specific AST parts, which allows the compiler to inline method calls even if the callee is a foreign function. Widening the compilation span across different languages enables the compiler to apply optimizations to a wider range of code.

Message resolution also allows Truffle's dynamic compiler to apply *escape analysis* and *scalar replacement* [31] to foreign objects. Consider a JavaScript program that allocates an object, which is used by the C part of an application. Message resolution ensures that Truffle's escape analysis can analyze the object access, 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.

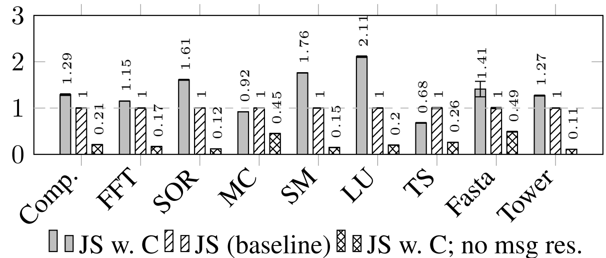


Figure 9: Performance evaluation of multi-language applications without message resolution (higher is better).

To demonstrate the performance improvement due to message resolution we temporarily disable it. When disabling message resolution, the $\mathrm{LI}_{\mathrm{Host}}$ still uses the dynamic access to access foreign objects but the TruffleVM does not replace the messages in $t_{a,m} \in T_{N^A \cup N^{\mathrm{Msg}}}$. It uses $\mathrm{LI}_{\mathrm{Foreign}}$ to locally execute the access operation and return the result. We do not introduce additional complexity to the dynamic access. However, $\mathrm{LI}_{\mathrm{Host}}$ has to treat $\mathrm{LI}_{\mathrm{Foreign}}$ as a black box, which introduces a language boundary. In Figure 9 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 that is for JavaScript and C on average $6\times$ slower than the performance with message resolution. The dynamic compiler cannot perform optimizations across language boundaries, which explains the loss in performance. We expect similar results for the other configurations, however, we have not measured them yet because disabling message resolution for an LI requires a significant engineering effort.

6. Foreign Function Interfaces with Truffle

We consider the TruffleVM to be very different from FFIs. Usually, an FFI composes a fixed pair of languages, while the TruffleVM allows interoperability between arbitrary languages as long as they comply with the requirements described in Section 2.

Of course, the dynamic access can also be used for implementing FFIs, which we discussed in previous work:

C data access for JavaScript [15]: In this work we accessed C data from within JavaScript, which is similar to using `Typed Arrays` in JavaScript. The performance evaluation shows that using C arrays from within JavaScript is on average 19% faster than using `Typed`

`Arrays` [15]. Performance improves because a C data access is more efficient than accessing a `Typed Array`.

C extension support for Ruby [17]: C extensions allow Ruby programmers to write parts of their application in C. The Ruby engine MRI exposes a C extension API that consists of functions to access Ruby objects from C. In [17] we composed Ruby and C by implementing this API in TruffleC. TruffleC substitutes every call to this API with the dynamic access that accesses Ruby data. We ran existing Ruby modules that used a C extension for the computationally intense parts of the algorithm, i.e., Ruby code calls C functions frequently, which in turn heavily access Ruby data. The performance is on average over $3\times$ faster than the MRI implementation of Ruby running native C extensions.

7. Related Work

7.1 Common Language Runtime

The Microsoft Common Language Infrastructure (CLI) describes LIs that compile different languages to a common IR that is executed by the Common Language Runtime (CLR) [6]. The CLR can execute conventional object-oriented imperative languages and the functional language F#. Languages running under the CLR are restricted to a subset of features that can be mapped to the Common Language Specification (CLS) of the shared object model, i.e., a language that conforms to this CLS can exchange objects with other conforming languages. CLS-compliant LIs generate metadata to describe user-defined types. This metadata contains enough information to enable cross-language operations and foreign object accesses.

The CLS cannot directly call low-level languages such as C. Native calls are done via the annotation-based PInvoke and the FFI-like IJW interface, which uses explicit marshaling and a pinning API.

Microsoft's approach is different from ours because it forces CLS-compliant languages to use a predefined representation of data on the heap and to use a shared set of operations to access it. The TruffleVM, on the other hand, allows every language to have its own representation of objects and to define individual access operations. Object accesses are done via the dynamic access, i.e., they are mapped to language-independent messages which are dynamically resolved for different languages at runtime. We embed foreign-language-specific accesses into the host IR, which creates a homogeneous IR that the dynamic compiler can optimize even across language boundaries.

We argue that the TruffleVM is more *flexible* because co-operating languages are not limited to languages that comply with a predefined object model and a fixed set of access operations. The TruffleVM allows the efficient composition of managed and unmanaged languages on top of a single runtime and the exchange of data as diverse as dynamic JavaScript objects and unmanaged C structures.

7.2 Foreign Function Interfaces

Most modern VMs expose an FFI such as Java's JNI [24], Java's Native Access, or Java's Compiled Native Interface. An FFI defines a specific interface between two languages: a pair of languages can be composed by using an API that allows accessing foreign objects. The result is rather inflexible, i.e., the host language can only interact with a foreign language by writing code that is specific to this pair of languages. Also, FFIs primarily allow integrating C/C++ code, e.g., Ruby and C (native Ruby extension), R and C (native R extensions), or Java and C. They hardly allow integrating code written in a different language than C.

Wrapper generation tools (e.g. the tool Swig [4] or the tool described by Reppy and Song [28]) use annotations to generate FFI code from C/C++ interfaces, rather than requiring users to write FFI code by hand. However, these tools add a maintenance burden: programmers need to copy API definitions and apply annotations outside the original source code. A similar approach is described in [23], where existing interfaces are transcribed into a new notion 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. [32] 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. [22] describe *invisible VMs*, which allow a simple and low-overhead foreign function interfacing and the direct use of native tools. They implement the Python language and minimize the FFI overhead.

There are many other approaches that target a fixed pair of languages [5, 12, 21, 29, 37]. These approaches are tailored towards interoperability between two specific languages and cannot be generalized for arbitrary languages and VMs. In contrast to them, the TruffleVM provides true cross-language interoperability rather than just pairwise interoperability: we can compose languages without writing boilerplate code, without targeting a fixed set of languages, and without introducing a compilation barrier when crossing language boundaries. The TruffleVM requires LIs to implement the dynamic access in order to become interoperable with other languages. Hence, the TruffleVM can be easily extended with new languages.

7.3 Multi-Language Source Code

Another approach to cross-language interoperability is to compose languages at a very fine granularity by allowing the programmer to toggle between syntax and semantics of languages on the source code level [2, 3, 18]. Jeannie [18] allows toggling between C and Java, hence, the two languages can be combined more directly than via an FFI. A similar approach is used by Barrett et al., in which the authors describe a combination of Python and Prolog called Unipycation [2]

or Python and PHP called PyHyp [3]. Unipycation and PyHyp compose languages by directly combining their interpreters. We share the same goals with Barrett et al., namely to retain the performance of different language parts when composing them, however, the TruffleVM is not restricted to a fixed set of languages.

Jeannie, Unipycation, and PyHyp allow a more fine-grained language composition than the TruffleVM. However, the code, written by programmers, consists of multiple languages, which requires adaption of source-level tools (including debuggers).

7.4 Interface Definition Languages

Interface Description Languages (IDLs) implement cross-language interoperability via message-based inter-process communication between separate runtimes. An IDL allows the definition of interfaces that can 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 [26, 30, 34]. 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 and its toolchain).

Using IDLs in the context of single-process applications has only been explored in limited ways [9, 35]. Also, these approaches retain the marshalling overhead and cannot share objects directly. The TruffleVM avoids copying of objects at language borders and rather uses the dynamic access. Message resolution makes language boundaries transparent to the dynamic compiler, which can optimize across language boundaries. Furthermore, by allowing the programmer to implicitly access foreign objects we make the mapping of language operations to messages largely the task of language implementers rather than the task of end programmers.

7.5 Multi-Language Semantics

The semantics of multi-language composability is a well researched area [1, 11, 12, 25, 33, 37], however, most of these approaches do not have an efficient implementation. Our work uses some inspiring ideas from existing approaches (such as *like types* from Wrigstad et al.[37], Section 4.1) and therefore stands to complement such efforts.

8. Conclusion

In this paper we presented a novel approach for composing language implementations, hosted by a shared runtime. The TruffleVM allows programmers to directly access foreign objects using the operators of the host language. Language implementations access foreign objects via the dynamic access, which means that language implementations use language-independent messages that are resolved at their first execution and transformed to efficient language-specific operations. We define a mapping from language-specific object accesses to language-independent messages and vice versa. This approach makes the mapping of language operations to messages largely the task of the language implementer rather than the task of the end programmer. The dynamic access allows us adding new languages to our platform without affecting existing languages.

The TruffleVM leads to excellent performance of multi-language applications because of two reasons: First, message resolution replaces language-independent messages with efficient 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.

The work presented in this paper improves the simplicity, the flexibility, and the performance of multi-language applications. It is the basis for a wide variety of different areas of future research on which we will focus. 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 paper. We thank Daniele Bonetta, Stefan Marr, and Christian Wirth for feedback on this paper. We especially thank Stephen Kell for significant contributions to our literature survey.
Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Proceedings of POPL*, 1989. URL http://doi.acm.org/10.1145/75277.75296.

[2] E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A Case Study in Cross-language Tracing. In *Proceedings of the 7th VMIL*, 2013. URL http://doi.acm.org/10.1145/2542142.2542146.

[3] E. Barrett, L. Diekmann, and L. Tratt. Fine-grained Language Composition. *CoRR*, abs/1503.08623, 2015.

[4] D. M. Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of USENIX Tcl/Tk workshop*, 1996.

[5] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C natively. *Electronic Notes in Theoretical Computer Science*, 2001.

[6] D. Box and C. Sells. Essential .NET. The Common Language Runtime, 2002.

[7] S. Brunthaler. Efficient Interpretation Using Quickening. In *Proceedings of DLS*, 2010. URL http://doi.acm.org/10.1145/1869631.1869633.

[8] D. Chisnall. The Challenge of Cross-language Interoperability. *Commun. ACM*, 2013. URL http://doi.acm.org/10.1145/2534706.2534719.

[9] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of ICFP*, 1999. URL http://doi.acm.org/10.1145/317636.317790.

[10] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. 1986. URL http://doi.acm.org/10.1145/5666.5673.

[11] K. Gray. Safe Cross-Language Inheritance. In *ECOOP*. 2008. URL http://dx.doi.org/10.1007/978-3-540-70592-5_4.

[12] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained Interoperability Through Mirrors and Contracts. In *Proceedings of OOPSLA*, 2005. URL http://doi.acm.org/10.1145/1094811.1094830.

[13] M. Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of SPLASH*, 2014. URL http://doi.acm.org/10.1145/2660252.2660256.

[14] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of PPPJ*, 2014. URL http://dx.doi.org/10.1145/2647508.2647528.

[15] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An Efficient Approach for Accessing C Data Structures from JavaScript. In *Proceedings of ICOOOLPS*, 2014. URL http://dx.doi.org/10.1145/2633301.2633302.

[16] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe Execution of C on a Java VM. In *Proceedings of PLAS*, 2015. URL http://doi.acm.org/10.1145/2786558.2786565.

[17] M. Grimmer, C. Seaton, T. Wuerthinger, and H. Moessenboeck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of MODULARITY*, 2015. URL http://doi.acm.org/10.1145/2724525.2728790.

[18] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface Developers Their Wishes. In *Proceedings of OOPSLA*, 2007. URL http://doi.acm.org/10.1145/1297027.1297030.

[19] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-typed Object-oriented Languages with Polymorphic Inline Caches. In *ECOOP*. 1991. . URL http://dx.doi.org/10.1007/BFb0057013.

[20] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of PLDI*, 1992. URL http://doi.acm.org/10.1145/143095.143114.

[21] S. P. Jones, T. Nordin, and A. Reid. Greencard: a foreign-language interface for haskell. In *Proc. Haskell Workshop*, 1997.

[22] S. Kell and C. Irwin. Virtual machines should be invisible. In *Proceedings of SPLASH*, 2011. URL http://doi.acm.org/10.1145/2095050.2095099.

[23] F. Klock II. The layers of larcenys foreign function interface. In *Scheme and Functional Programming Workshop*. Citeseer, 2007.

[24] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Boston, MA, USA, 1st edition, 1999.

[25] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of POPL*, 2007. URL http://doi.acm.org/10.1145/1190216.1190220.

[26] M. D. Network. XPCOM Specification. https://developer.mozilla.org/en-US/docs/Mozilla/XPCOM, 2014.

[27] Oracle. OpenJDK: Graal project. http://openjdk.java.net/projects/graal/, 2013.

[28] J. Reppy and C. Song. Application-specific Foreign-interface Generation. In *Proceedings of GPCE*, 2006. . URL http://doi.acm.org/10.1145/1173706.1173714.

[29] J. R. Rose and H. Muller. Integrating the scheme and c languages. In *Proceedings of LFP*, 1992. URL http://doi.acm.org/10.1145/141471.141559.

[30] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 2007.

[31] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of CGO*, 2014. . URL http://doi.acm.org/10.1145/2544137.2544157.

[32] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining Java Native Calls at Runtime. In *Proceedings of VEE*, 2005. URL http://doi.acm.org/10.1145/1064979.1064997.

[33] V. Trifonov and Z. Shao. *Safe and principled language interoperation*. 1999.

[34] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the CORBA Component Model. In *Component-Based Software Engineering*, 2001.

[35] M. Wegiel and C. Krintz. Cross-language, Type-safe, and Transparent Object Sharing for Co-located Managed Runtimes. In *Proceedings of OOPSLA*, 2010. URL http://doi.acm.org/10.1145/1869459.1869479.

[36] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of PPPJ*, 2014. URL http://dx.doi.org/10.1145/2647508.2647517.

[37] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of POPL*, 2010. . URL http://doi.acm.org/10.1145/1706299.1706343.

[38] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of DLS*, 2012. . URL http://doi.acm.org/10.1145/2384577.2384587.

[39] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of ONWARD!*, 2013.