Supporting Language Interoperability by Dynamically Switched Behaviors

Jan Kurš[1] Jan Vraný[1] Alexandre Bergel[2]

[1]Software Engineering Group,
Faculty of Informatics,
Czech Technical Universtity in Prague
{kursjan, jan.vrany}@fit.cvut.cz

[2]Pleiad Lab, Department of Computer Science (DCC)
University of Chile, Chile
http://bergel.eu

Abstract. Software programs are often written in more than one programming language as the emergence of domain specific languages testifies. Language interpreters are easily embeddable and performances are usually satisfactory. However, inter-language interaction remains a field tarnished by poor performances. The reason is that alien objects are wrapped, implying the use of expensive forwarding and converting mechanism.
We propose to represent alien objects as the set of different states and behaviors it may have by moving between languages, thus avoiding wrapping and conversion. We have validated our solution on integration of Java and Smalltalk programming languages.

Keywords: Programming Language, Virtual Machine, Object Transitions, Java, Smalltalk

1 Introduction

The last decade has seen the advent of domain specific languages and support for multi languages. Common execution platforms, including the JVM and .Net, are nowadays fit to execute programs written in more than one programming language. Whereas the execution mechanism needed to interpret these languages are fairly well accepted [7, 10], the way languages interact and exchanges values still remains an open topic.

The large majority of embedded languages convert or wrap objects when they cross the language boundary [12]. When an object is passed from one language interpreter to another, it is either converted or wrapped: values like integers, floats, booleans, characters, and strings are merely converted while the remaining objects are simply wrapped.

Whereas objects conversion and wrapping is a globally accepted among domain specific languages and scripting languages, it is the source of several problems and limitations. Consider a plain Java dictionary produced by a Java program. This dictionary is represented as an instance of java.util.HashMap. A JRuby interpreter will consider this object as a wrapped alien object. All calls done on this object implies a conversion or

wrapping of its arguments and a delegation by the wrapper to the real objects. Sending the JRuby message put("One", 1) to the Java dictionary converts the JRuby string "One" and the JRuby integer 1 into their corresponding Java values. Delegating messages has a cost which is significant when intensively use.

A second problem is about object identity. When this Java dictionary is passed a second time to JRuby, it has to be wrapped using the same wrapper that was used for the first time. A bijective mapping between alien objects and wrappers has to be enforced. The wrapper used the second time has to be physically the same than the first wrapper (i.e., having the same pointer). Again, this comes at a fairly high cost in case of intensive object passing.

Instead of representing aliens objects as a wrapper in the host language, we propose to extend the definition of an object as a set of contextualized variable layouts and behavior definition.

The proposed approach has been validated on Smalltalk/X programming environment that runs code in Smalltalk and Java programming languages. We have modified metaobject protocols in Smalltalk/X in order to implement proposed approach efficiently.

The paper is organized as follows: The Section 2 introduces simple code and describes problems caused by language interaction. Our solution is described in Section 3 and the implementation outlined in Section 4. The Section 5 discusses, how our approach solves the problems from the Section 2 and what are the limitations of our approach.

2 Problem

2.1 Example

Consider code in Figure 1 and Figure 2 that demonstrates interaction of Java and Smalltalk languages. In Figure 1, there is a method `sayHello` that selects language according to the locale and print appropriate greeting. The `sayHello` method expects a map with translations to be passed as a parameter. In Figure 2, there is a Smalltalk code that prints greeting using the Java code shown in Figure 1. The Smalltalk creates a translations as an instance of class `Dictionary` and then invokes `sayHello` method to print the greeting.

During the invocation of `sayHello` method on an instance of `MultilanguageHelloWorld` class, an instance of Smalltalk `Dictionary` is passed to Java method that expects an instance of `java.lang.Map`. In that case, we say that the Smalltalk object *crossed the language boundary*.

2.2 Problem description

The problem is that the `Dictionary` cannot be used as parameter of `sayHello` method directly – it is a different object from completely different type hierarchy with different set of methods. Yet we intuitively feel, that the `Dictionary` is Smalltalk equivalent of `java.util.Map`. Both are used to store values under arbitrary key.

```
public class MultilanguageHelloWorld
{
    public void sayHello(HashMap dictionary)
    {
        String key = getLocale().getLanguage();
        System.out.println(dictionary.get(key));
    }
}
```

Fig. 1. Java class `MultilanguageHelloWorld` that can print greetings according to the locale.

```
greetings := Dictionary new
    at: 'en' put: 'Hello World';
    at: 'cs' put: 'Ahoj světe'.

MultilanguageHelloWorld new
        sayHello(greetings).
```

Fig. 2. Smalltalk code interacting with Java object - `MultilanguageHelloWorld`

If we want to let Java and Smalltalk code from Figure 1 and Figure 2 interact smoothly, there are two basic approaches; First, do not create an instance of `Dictionary` – create an instance of `java.lang.HashMap` class from the very beginning. Or second, if necessary, create a new `HashMap` object and copy data from the `Dictionary` to the `HashMap`.

In the first case, there may arise problem when the object creation is not under our control. For example, if the translation mapping is obtained from a third-party library which cannot be modified. The second approach is time consuming and has higher memory requirements. We have to take care about the object identity as well: if we created a new object every time the object is passed from Smalltalk to Java, multiple Java `HashMaps` would represent the same Smalltalk `Dictionary`. Moreover, data should be kept in sync: if something changed in the Java `HashMap`, we should update the Smalltalk `Dictionary` object.

Usage of a proxy [6] object is third, more advanced approach. The proxy eliminates problems with data synchronization. Nevertheless problems with identity remains and we have to map proxies to their subjects which causes extra performance and memory overhead.

3 Our solution

As mentioned before, our approach represents single object in various languages by the only one physical object with dynamically changed behaviour. Object behaviour in specific language is described by a structure that we call *behaviour object*. In other words, the *behaviour object* describes behaviour of given object in scope of given language. In

most of languages, the behaviour object is its class, in prototype languages the behavior object might be represented by object map [3]. Any physical object may be associated with as many behaviour objects as is the number of languages in which the object is used. Whenever an object crosses language boundary we dynamically change a behaviour object according to the actual language. The `greetings` object from Figure 2 would have two behaviour objects associated – one for smalltalk language representing the `Dictionary` class and one used in Java representing the `java.util.Map`.

Next important part of our approach is a mapping of an object state. We will call an ordered set of object fields as *an object layout*. *A primary object layout* is then an object layout defined by the language where the object was instantiated. We will call a set of object fields and their respective values as *an object state* – an object state is an object layout with values. Similarly, *a primary object state* is a state of the object with primary layout. Any method in any language may change an object state. Unfortunately, each behaviour object may require different object layout. Because we share the same physical object among languages, a mapping function has to map primary object state to desired object state and vice versa.

The idea of shared behaviour and mapped state is depicted in Figure 3. In the upper left-hand corner there is a physical object `java.lang.String` composed of behaviour and state. In the upper right-hand corner there is a similar structure for Smalltalk `String`. In the bottom, there is a composed object – one physical object with both, Smalltalk and Java behaviour. The behaviour is simply added, the state has to be mapped from Smalltalk to Java.

We will describe our approach more formally now. Let's have a virtual machine VM which is able to interpret native language L_1 and alien language L_2. Let's have a program P_1 written in L_1 and a program P_2 written in L_2. P_2 interacts with P_1. As an input parameter, P_1 expects an object O_1 with behaviour described by a behaviour object B_1. P_2 creates an object O_2 with a primary layout $A_2 = f_{21}, f_{22}, \ldots, f_{2n}$ and with behaviour described by a behaviour object B_2. The object layout A_2 with values is an object state S_2. B_2 differs from B_1. B_1 expects an object to have a layout $A_1 = f_{11}, f_{12}, \ldots, f_{1m}$. The object layout A_1 with values is an object state S_1. We want to use O_2 in P_1 as O_1. An example could be find in Figure 1, Figure 2 and described in Section 2.1.

We need to define mapping from S_2 to S_1. Such mapping has to satisfy two requirements. First, an appropriate value has to be determined from S_2 when the value of a field $f \in A_1$ is needed. Second, S_2 has to be updated accordingly when a field $f \in A_1$ is being set. If the layouts of A_1 and A_2 are identical, the mapping is trivially identity mapping. If it is not possible to map S_2 to S_1, O_1 and O_2 could not be considered to be equivalent in L_1 and L_2 – they have too little in common. In the rest of cases, the mapping has to be specified explicitly.

It is also necessary to provide mapping that maps languages and behaviour object, i.e., that the object O_1 with behaviour of B_1 in language L_1 will be associated with behaviour B_2 in language L_2 and vice versa. Whenever a message is sent to O_2 from L_1 (L_2 respectively), a message selector will be looked up in B_1 (B_2 respectively). During a program execution, various situations may occur:

- When a method is called on O_2 from P_1, the method is looked up in B_1.

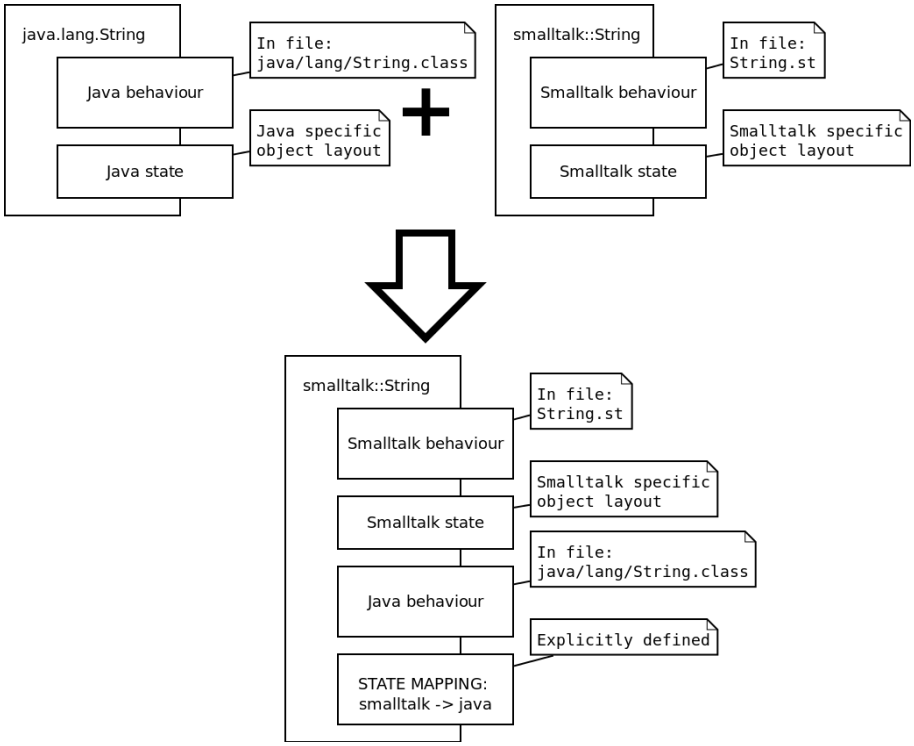

Fig. 3. One physical object with multiple behaviours (in the bottom) is composed from `java.lang.String` and Smalltalk `String` (in the top). The behaviour is added, the state is mapped.

- When a field $f \in A_1$ of O_1 is being read from P_1 and A_2 is the primary object layout, then the mapping from S_1 to S_2 is used to compute the value of f based on S_2.
- When a field $f \in A_1$ of O_1 is being set from P_1 and A_2 is the primary object layout, the mapping from S_1 to S_2 is used and the S_2 is updated.
- When the object O_2 is passed from P_2 passed to P_1 (O_2 crosses the language boundary), B_1 is assigned to O_2.
- When the object O_2 is passed from P_1 back to P_2, B_2 is assigned to O_2 again.

4 Implementation

We have validated our solution on Java and Smalltalk programming languages. We use Smalltalk/X virtual machine to interpret Java and Smalltalk language. We employ metaobject protocol [11] that is implemented in Smalltalk/X VM [14] to change method and field lookup semantics.

We use standard Smalltalk class as a behaviour object for Smalltalk objects. We use special Smalltalk object similar to Java class as a behaviour object for Java objects.

Essentially, some objects have two classes – one for Smalltalk and second for Java. We modified method lookup in order to reflect an existence of multiple behaviour objects per object as follows:

```
Lookup>>lookupMethodForSelector:selector
        for:receiver
        withArguments:argArrayOrNil
        context: context
    | behaviour |
    behaviour := receiver behaviourObjectFor: context language.
    behaviour lookupMethodForSelector: selector
              withArgumets: argArrayOrNil.
!
Object>>behaviourObjectFor: language
    ^ ObjectRegister instance
        getCorrespondingClassOf: self class
        inLanguage: language.
!
Object>>primaryBehaviourObject
    ^ self class
!
```

Furthermore we modified field accessor functions to be able to apply mapping between different states as follows:

```
Lookup>>getFieldForFieldName:fieldName
        for:receiver
        context: context
    | behaviour primaryBehaviour |
    behaviour := receiver behaviourObjectFor: context language.
    primaryBehaviour := receiver primaryBehaviourObject.
    ^ StateMapping instance
        getFieldNamed: fieldName
        fromBehaviour: behaviour
        toBehaviour: primaryBehaviour
        forObject: receiver.
!
```

Last but not least, we introduced global map, where the equivalent types may be registered together with the state mapping functions as follows:

```
ObjectRegister>>addBehaviour: behavirouObject
                to: primaryBehaviourObject
    | behaviourObjectCollection |
    behaviourObjectCollection := self at: primaryBehaviourObject.
    behaviourObjectCollection add: behavirouObject.
!
```

A demonstration of our solution's abilities is depicted in Figure 4 and Figure 5. Equivalent codes and outputs will in other languages will be described later, in Section 6 which compares our implementation with another ones.

```
SOURCE:                              OUTPUT:
string := 'Smalltalk string'.
smalltalkInfo info: string.          info from Smalltalk world
    // string class                      class: String class
    // string hash                       hash: 197479768
javaInfo info: string.               info from Java world
    // string.getClass()                 class: java.lang.String
    // string.hashCode()                 hash: 7110656
java equals: string and: string      object equals:
    // string1 == string2                true
```

Fig. 4. An interaction of Smalltalk String with Java code.

The Figure 4 is divided into two parts. There is a source in the left and output in the right. The `smalltalkInfo`'s method `info` prints a class and hash code of a parameter. The `javaInfo`'s method `info` prints a class and hash code of a parameter as well – but it is written in Java. The `javaInfo`'s method `equals` compares identity of parameters and prints `true` if objects are identical, `false` otherwise. As you can see, the String object has appropriate class and hash in both of the languages.

```
SOURCE:                              OUTPUT:
set := HashSet new with: 1
                   with: 6.
smalltalkInfo info: set.             info from Smalltalk world
                                         class: HashSet class
// info(Object o)                        hash: 6537216
javaInfo info: set.                  info(Object o) from Java world
                                         class: java.util.HashSet
//info(Set s)                            hash: 6537216
javaInfo infoSet: set.               infoSet(Set s) from Java world
                                         class: java.util.HashSet
//info(Set s)                            hash: 6537216
javaInfo infoHashSet: set.           infoHashSet(HashSet s) from ...
                                         class: java.util.HashSet
                                         hash: 6537216
```

Fig. 5. Intraction of Smalltalk object with Java code.

The Figure 5 is divided into two parts as well – source in the left and output in the right. The `smalltalkInfo`'s method `info` is the same as in Figure 4. It prints class and hash code of a parameter. The `javaInfo`'s method `info(Object o)` prints a class and hash code of a parameter – it demonstrates that Smalltalk object may be handled as Java object even though `java.lang.Object` is not anywhere in Smalltalk class hierarchy. The `javaInfo`'s method `info(Set s)` demonstrates that Smalltalk

object may be handled as Java interface. The `javaInfo`'s method `info(HashSet s)` demonstrates that Smalltalk object may be handled as ordinary Java class.

5 Discussion

In case an object is shared between multiple languages and its behaviour is dynamically changed according to the actual language, following problems are naturally solved:

Object identity The object identity is based on an object pointer comparison. Since we represent objects by the same pointer in computer memory, no problem arises.

Explicit copy If there is no support for automatic object conversions between, programmers have to take extra care while passing object across the language boundary. It may happen that an alien object with inappropriate behaviour will be used that may rise an exception. The error may be prevented by explicit call of a conversion method. On the other hand, if the behaviour is changed automatically, the work with alien objects is transparent – they look like native objects. No extra care has to be taken while passing object across the language boundary.

Data synchronization If objects has to be copied while crossing the language boundary, synchronization of data has to be ensured. Our solution work with the same data so it is not a deal any more.

Memory overhead Object copy implies memory overhead since all data are duplicated. Proxy objects may be light-weighted as to not consume too much memory, nevertheless due to necessity to preserve an identity, an extra memory is consumed by (global) mappings of objects to their respective proxies. Such a mapping is not only memory consuming but also requires proxies to be weak-referenced. Weak references affects garbage collector performance since all weak references must be treated specially. In our solution, objects are shared between multiple languages and so the memory is not occupied redundantly. Behaviour objects does not cause any memory overhead as well, since they are already present in particular languages.

Questions regarding the reflective facilities may arise.

Object class Object class could be obtained by sending appropriate method (`class` in Smalltalk, `getClass()` in Java). The return value is metaobject which keeps information about methods, fields, subclasses, super class and others. It could be said that the return value is the behaviour object (in some form) currently associated with the given object. Our technique does not affect this functionality. For each language, appropriate object representing the class is returned. From the point of any particular language, an object has one class.

Object superclass Object superclass is stored in its behaviour object. Since the correct behaviour object is always returned, asking it for a superclass will return a corresponding superclass in scope of given language.

Super sends Since the problems has not arose in previous case, it is not problem to invoke super send. Nevertheless if Y_2 `extends` X_2 in language L_2 (with object layouts A_{Y2} and A_{X2}) and Y_1 exists in language L_1 (with object layout A_{Y1}) and Y_1 is used in language L_2 as Y_2, the Y_1 must provide mapping from A_{B1} to $A_{B2} \cup A_{A2}$. In other words, Y_1 must provide mapping to the complete object layout of Y_2 – including superclasses.

5.1 Implementation limitations

There are several possible implementations of our approach. We have chosen to profit from metaobject protocol implemented in Smalltalk/X as described in Section 4. Another suitable metaobject protocol is provided by Dynamic Language Runtime [5] framework built on top of Common Language Runtime [13]. Unfortunately, it is not possible to integrate C# and IronRuby [2] or IronPython [1] this way, because existing C# does not use Dynamic Language Runtime.

In Smalltalk, another techniques like `doesNotUnderstand:` hook and Java bytecode instrumentation could be used. The `doesNotUnderstand:` hook allows method lookup customization, but this technique negatively influences performance. The bytecode instrumentation may be used to replace method call in bytecode by another routine in bytecode that takes multiple behaviour objects into the account. The get field and set field bytecode instructions may be replaced by similar routine that take state mapping into the account.

6 Related Work

6.1 JRuby

JRuby [4] is an implementation of Ruby language running on top of Java Virtual Machine. Generally, JRuby objects may interact with Java code. Nevertheless there are some "pain points". In case of Strings, they may be shared between JRuby and Java without any limitations. A class of a String object changes appropriately, a hash code is computed correctly and an identity is preserved. This demonstrates code depicted in Figure 6.

```
SOURCE:                          OUTPUT:
string = "ruby string"
rubyInfo.info(string)            info from Ruby world
    // string.class                  class: String
    // string.hash                   hash: 250737224
javaInfo.info(string)            info from Java world
    // string.getClass()             class: java.lang.String
    // string.hashCode()             hash: 916834583
javaInfo.equals(string, string)  object equals:
    // string1 == string2            true
```

Fig. 6. Interaction of Ruby String with Java code.

The code in Figure 6 is written in Ruby which interacts with Java. The code is equivalent to the code in Figure 4 which is written in Smalltalk. Regarding strings, there is no difference between abilities of JRuby and our solution.

Generally, JRuby objects may be used as a parameter whenever the parameter is `java.lang.Object` because JRuby objects inherit from `java.lang.Object`.

Moreover, JRuby object may be used as a parameter of Java method in case the parameter is Java interface and the Ruby object implements the interface. Yet, if a Java method expects standard object (subtype of `java.lang.Object`), exception is raised. This demonstrates a code depicted in Figure 7.

```
set = Set[1, 3, 4, 11]
rubyInfo.info(set)

                                  info from ruby world
                                      class: Set
// info(Object o)                     hash: 24118174
javaInfo.info(set)                 info(Object o)
                                      class: org.jruby.RubyObject
// infoSet(Set s)                     hash: 24118174
javaInfo.infoSet(set)              infoSet(Set s)
                                      class: ....InterfaceImpl
// infoHashSet(HashSet hs)            hash: 21279119
javaInfo.infoHashSet(set)          infoHashSet(HashSet hs)
                                      cannot convert class
                                      org.jruby.RubyObject to
                                      java.util.HashSet
```

Fig. 7. Interaction of Ruby object with Java code.

The code in Figure 7 is written in Ruby which interacts with Java. The code is equivalent to the code in Figure 5 which is written in Smalltalk. Source is in the left, output is in the right. As you can see, JRuby allows to pass Ruby object to methods, which expects `java.lang.Object` and Java interface, but not `HashSet`. Our implementation allows to pass Smalltalk object to any of the methods.

6.2 Jython

Jython [9] is an implementation of Python language running on top of Java Virtual Machine. Jython objects may interact with Java code, but there are some "pain points" as well. In case of Strings, they may be shared between Jython and Java without limitations. A class of an object is changed appropriately, a hash code is computed correctly and an identity is preserved. This demonstrates code depicted in the Figure 8.

There is a code written in Jython which interacts with Java objects in the Figure 8. The code is code equivalent to the code in Figure 4 which is written in Smalltalk. Source is in the left, output is in the right. Again, regarding strings, there is no difference between abilities of Jython, JRuby and our solution.

Yet, it is not easy to use Jython object as parameter of Java method. There is a mechanism called *Object Factory* in the Jythonbook [8] but it requires lots of code overhead. The mechanism cannot be used in all use cases. Generally, it is not possible to pass Jython's `ImmutableSet` instance into the Java method expecting either

```
SOURCE:                              OUTPUT:
string = "jython string"
jythonInfo.info(string)              info from Jython world
    // string.__class__                  class: <type 'str'>
    // string.__hash__()                 hash: 1857618127
javaInfo.info(string)                info from java world
    // string.getClass()                 class: java.lang.String
    // string.hashCode()                 hash: 1857618127
javaInfo.equals(string, string)      object equals:
    // string1 == string2                true
```

Fig. 8. Interaction of Jython String with Java code.

`java.util.Set` or `java.util.HashSet`. This is a difference between our solution and Jython, since our solution is not limited in these use cases.

7 Conclusion

In this paper we have presented a dynamic behaviour switching mechanism to support language interoperability. When an object is passed from one programming language to another, its behaviour is dynamically switched to what the other language expects, allowing programmers to work with alien objects in a natural way. The same physical object is used in all languages, therefore there is no runtime overhead caused by copying objects and by maintaining object identity. A mapping from class in one language to corresponding class in the other language is provided by user as well as a mapping of object state.

References

1. IronPython, August 2010. `http://ironpython.net/`.
2. IronRuby, August 2010. `http://ironruby.net/`.
3. Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF — a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 49–70, October 1989.
4. Charles Nutter et. al. JRuby Project, August 2010. `http://jruby.org/`.
5. Bill Chiles and Alex Turner. Dynamic Language Runtime, August 2010. `http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs`.
6. Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

7. Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection – a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006.
8. Josh Juneau, Jim Baker, Victor Ng, Leo Soto, and Frank Wierzbicki. Jython Book v1.0 documentation, March 2010.
`http://www.jython.org/jythonbook/en/1.0/`.
9. Jython, February 2011.
`www.jython.org`.
10. Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *PPPJ'08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 189–197, Modena, Italy, 2008. ACM.
11. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
12. Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007.
13. E. Meijer and J. Gough. Technical overview of the common language runtime, 2000.
14. Jan Vraný. Supporting multiple languages in virtual machines. Dissertation thesis, Czech Technical University, December 2010.