Benzo: Reflective Glue for Low-level Programming

Camillo Bruni Stéphane Ducasse
Igor Stasenko
RMoD, INRIA Lille - Nord Europe, France
http://rmod.lille.inria.fr

Guido Chari
Departamento de Computación, FCEyN, UBA
and CONICET
http://lafhis.dc.uba.ar

Abstract

The goal of high-level low-level programming is to bring the abstraction capabilities of high-level languages to the system programming domain, such as virtual machines (VMs) and language runtimes. However, existing solutions are bound to compilation time and expose limited possibilities to be changed at *runtime and from language-side*. They do not fit well with fully reflective languages and environments.

We propose Benzo[1], a lightweight framework for high-level low-level programming that allows developers to generate and execute at runtime low-level code (assembly). It promotes the implementation, and dynamic modification, of system components with high-level language tools outperforming existing dynamic solutions.

Since Benzo is a general framework we choose three applications that cover an important range of the spectrum of system programming for validating the infrastructure: a Foreign Function Interface (FFI), primitives instrumentation and a just-in-time bytecode compiler (JIT). With Benzo we show that these typical VM-level components are feasible as reflective language-side implementations. Due to its unique combination of high-level reflection and low-level programming, Benzo shows better performance for these three applications than the comparable high-level implementations.

Categories and Subject Descriptors D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

Keywords system programming, reflection, managed runtime extensions, dynamic native code generation

1. Introduction

High-level low-level programming [16] encourages to use high-level languages such as Java to build low-level execution infrastructures or to do system programming. Frampton et al. present a framework that is biased towards a statically typed high-level language, taking strict security aspects into account. It is successfully used in experimental high-level self-hosted virtual machines (VMs) such as Jikes [3].

The results presented by Frampton et al. are inspiring for high-level system programming developers. Their approach promotes to tackle low-level system programming tasks with the tools and abstractions of high-level languages. However, the solution has certain limitations when applied to a dynamic and reflective context.

By the term "dynamic and reflective" we refer to combined reflective capabilities for a language to inspect (introspection) and change its own execution (intercession) at runtime [24].

The most important limitation we found when approaching dynamic high-level low-level programming attaining to the existent solutions is the following:

It is not possible to generate ad-hoc native code (assembly) at runtime and execute it dynamically.

We illustrate it with the following use case.

1.1 Use Case: Dynamic Primitive Instrumentation

To illustrate this limitation we take the example of dynamically intercepting VM primitives. In most managed runtimes, VM primitives are used for essential tasks such as object creation or provide fundamental functionality that can not be obtained otherwise at language-side [17, page 52].

Already the simple example of measuring the time spent in an essential primitive while executing a performance critical task is difficult to do efficiently.

Reflective solutions at language-side could take advantage of the intercession capabilities that allow changing or augmenting almost any behavior at runtime. In practice, this does not work for primitives. In addition, reflectively measuring the duration of the time primitive itself will easily cause meta-recursion. Hence in general reflective instrumentation will not work on primitives used for the instrumentation itself. Thus efficiently instrumenting VM primitives in a dynamic and reflective environment is not feasible in many cases.

Naturally if we leave the high-level realm there are efficient tools at hand for instrumentation. Existing solutions for efficient low-level instrumentation such as DTrace [12] work by installing static hooks. Though in a reflective language assumptions can change at runtime and thus static solutions are not appropriate. At the same time we clearly see that such instrumentation is not a typical high-level application. To combine these two worlds we need a different approach.

[1] The name Benzo originates from Benzocyclobuten which is an organic glue used in wafer production.

Approaching Dynamic Primitive Instrumentation with Benzo. Using Benzo framework for *reflective low-level programming* we show in Section 4.2, as one of three proof of concepts, a solution to the problem of efficient dynamic VM primitive instrumentation. By dynamically generating and activating native code from language-side we are able to create customized primitives. With Benzo even essential primitives can be dynamically change without the need of a system restart.

Although this is a clear example, Benzo is a general reflective high-level low-level programming framework that overcomes also other system programming limitations. We illustrate its advantages with other distinct examples such as a Foreign Function Interface (FFI) and a just-in-time bytecode compiler (JIT) in Section 4.

1.2 Bridging Abstraction Layers

Extending high-level language runtimes is difficult due to their static low-level construction which usually shares little resemblance with the language-side. Yet for tasks, like the previously presented primitive instrumentation, if we want to tackle it with a high-level language, we need solid low-level interaction.

Requirements. In Section 2 we describe the solutions offered by traditional approaches to modify or extend a language runtime: language-side libraries, reflective capabilities, VM extensions or hybrid approaches. However, none of them is powerful or general enough to support our use-case. A general and uniform solution is needed that spans over several abstraction layers. It has to interact on a high-level with the reflective capabilities of the language runtime and at the same time provide an interface to interact with low-level code. To stay flexible and compatible enough the solution should add these new key features with as little static low-level intrusion as possible.

- It must be *reflective* in the sense it must support *dynamic* changes of the language runtime (VM) without requiring a system restart.
- It should imply minimal changes to the existing low-level runtimes to *considerably reduce development efforts.*

Benzo a Framework for Reflective Low-level Programming. High-level low-level programming is a powerful technique for system programming without resorting to static low-level environments [16, 34] that almost fulfills our requirements. However in a reflective setup it fails to comply with the first requirement mentioned in the previous paragraph.

Our approach consists of Benzo, a lightweight, dynamic and reflective framework that tackles the exposed limitations. Benzo dynamically generates native code from language-side and can execute the changes in place. It relies only on a small set of generic VM extensions described in Section 3.1.

Framework applications. In Section 4 we advocate the contribution of this approach by providing three different incremental examples that heavily use the framework from language-side. They rely on it for extending or even improving language runtime capabilities. They consist of:

FFI A complete language-side Foreign Function Interface (FFI) implementation, described in Section 4.1.

Dynamic Primitives A language-side compilation toolchain that replaces system primitives at runtime with customized code, described in Section 4.2.

Language-side JIT Compiler A JIT compiler that works at language-side and interacts with the VM for code synchronization, described in Section 4.3

As illustrated by these three distinct examples, the contributions of this paper are:

- Encouraging the extension of high-level language runtimes through the use of reflective low-level programming promoting an open interaction with the low-level world without the overheads imposed by high-level one.
- A proof of concept of the proposal with the implementation and description of three different tools that heavily use reflective low-level programming and covers distinct scenarios.

2. Current Approaches for Modifying/Extending Runtimes

We present now an overview of the approaches used to extend a language runtime and expose their limits.

High-level languages are in general sustained by a VM and a vast set of libraries written in the language itself. Extending or improving the existing Runtimes is a difficult task. In most cases the VM is considered as a black box. Additionally the VM is written in a completely different language using another abstraction level than the one it supports. Typically high-level language VMs are written in C or C++. To address runtime extensions in this context there exist some known approaches:

Language-side Library based on implementing a new or existing library.
Language-side Reflective Extension relying on reflective features of the language.
VM Extension by writing plugins or changing the core of the VM.
Hybrid Extension by accessing external libraries using FFI.

The relation between the side concerning the abstraction and implementation levels (VM vs. Language) of these extensions is illustrated in Figure 1.

2.1 Language-side Library

The most straight forward solution for extending a language is to write libraries within the language itself. This option provides the advantage that the aggregate behavior is accessible and evolvable for any language developer.

However language-side libraries are constrained by the underlying managed runtime. The VM separates the language from the low-level internal details. As a consequence language-side libraries are not feasible for all feature requirements. For instance the previously mentioned example of instrumenting the runtime is not possible as a standard language-side extension without a considerable performance loss. So, even though we prefer extensions and optimizations at language-side, there are certain limitations of a managed runtime that can not be circumvented. If all language-side optimization opportunities have been exhausted it is exposing the need to resort to lower level approaches.

Language-side libraries are constrained to the capabilities of the underlying VM and thus not general enough. Addi-

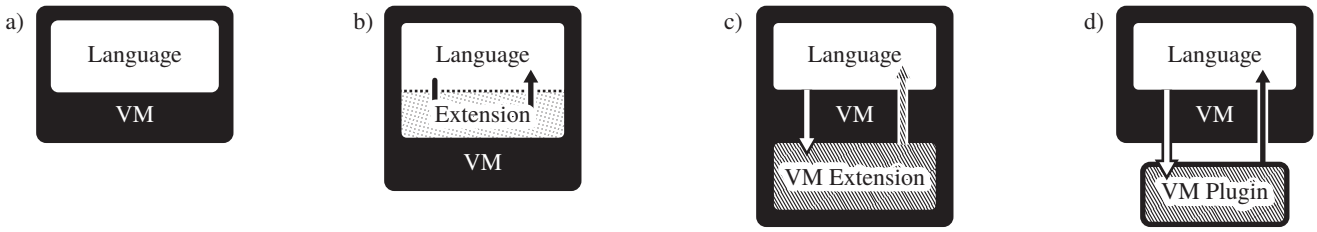

Figure 1: Comparing different extension mechanisms: a) language running on a standard VM, b) language-side implementation of an extension c) language using features from a VM extension, d) language using features from a VM plugin.

tionally not all performance bottlenecks can be addressed at language-side.

2.2 Language-side Reflective Extensions.

This is a subcase of the previous approach but in the context of reflective environments that expose particular characteristics.

For instance, Meta Object Protocols (MOP) [22] based on reflection [24] are used to define certain control points in the system to change the language. By composing meta objects it is possible to even modify the semantics of the language. Several languages such as Smalltalk, Python, and others provide reflective capabilities with different depths [4, 15, 32].

However most modern programming languages only have very limited support for intercession. Hence the possibilities for dynamically changing language semantics or features are limited. Furthermore reflective capabilities are hard to implement efficiently. Reflection imposes substantial performance penalties on most computations by postponing bindings [25].

Nevertheless there are exceptions for a subset of reflective behavior which are implemented efficiently using a high-level MOP [33]. Though these approaches remain as a few exceptions. In the typical low-level VM it is difficult to gain reflective access to language-side objects.

Similar to the previous case, our goal is to extend language features in a general way and it was shown that this is only partially possible by reflective extensions.

Reflective capabilities are not enough for general extensions. Even when suitable, they usually pose a significant performance overhead up to the point where they become unfeasible.

2.3 VM Extensions

Another approach is to attach plugins to the VM. Plugins are direct bindings to external libraries described at VM-side or libraries linked to the VM executable [8, Ch. 5]. They provide a performance boost in comparison to pure language-side solutions. Using highly optimized native libraries it is straightforward to outperform code written at language-side.

However, plugins are commonly written in the same language as the VM, at a low abstraction level. Few exceptions are self-hosted languages [29, 31, 34]. To support a fluent development process, VMs should come with an infrastructure for building extensions at same abstraction level than the language. Instead they tend to be very complex and to have sluggish building processes. For example, only a few VMs have high-level debugging facilities [19, 31, 34]. Also from a VM maintenance point of view, extensions have to be avoided if possible and should only be used for critical performance issues that can not be properly addressed at language-side. An example of how the complexity of the VM can affect development efforts is the core of the Self VM [30]. After reduced development resources parts of the complex but efficient compiler infrastructure had to be abandoned in favor of a more maintainable code-base.

VM extensions provide good performance but imply resorting to low-level tools where abstraction advantages of high-level languages are restricted.

2.4 Foreign Libraries

The last approach is to reuse an existing library usually implemented in a foreign language. The languages interact through a well-defined Foreign Function Interface (FFI). FFI-based extensions are an hybrid approach between pure language-side extensions and VM-side ones. Interaction with native libraries is supported by a dedicated VM functionality for calling external functions. This allows for a smooth interaction of external code and language-side code. FFI based extensions share the benefits of a maintainable and efficient language-side library with modest implementation efforts.

However, FFI is only a bridge or interface for allowing the interaction of different languages. It is not possible to directly synthesize new native features from language-side. For this purpose we have to interact with a custom-made native library. From an extension point of view this is close to the VM extensions discussed previously.

Additionally to the interface limitations, there exists a performance overhead in FFI for making the interaction between different languages possible. This is due to marshalling arguments and types between both languages [14, 28].

FFI allows developers to cross language-barriers with less effort than a VM extension and enables a tight integration with existing libraries. However, it is only an interface and depends on extensions already available. Moreover, performance penalties are considerable in some cases.

2.5 Summary

The approaches discussed above rely on language-side code with the exception of VM extensions. However we have shown limitations for all of them. Hybrid solutions such as FFIs are the only ones that come close to meet all our requirements. The only downside of FFIs is that it is not possible to directly synthesize new custom native functionality with them. Hence, for our purpose FFIs are not general enough as it is not possible to solve our initial use case for dynamically instrument primitives.

Benzo takes the advantages from the presented approaches but avoids their weaknesses. By using Benzo high-level developers tackle the problems in a uniform way by exploiting the debugging and development facilities provided by the language. Developers model their applications or libraries with a high-level language and have a clear interface for generating and executing efficient low-level code when needed, but inside the same language and with the same abstractions and reflective capabilities. In Section 5.2 we show that we achieve the performance requirements of low-level environments.

3. Benzo Implementation in a Nutshell

This section covers the necessary changes to make Benzo compatible with the Pharo VM and the language-side behavior contributions.

3.1 VM Context

Pharo is a Smalltalk dialect that emerged from the Squeak project [19]. The Pharo VM implementation [27] also evolved from the original Squeak bytecode interpreter. The current VM uses a moving Garbage Collector (GC) with two generations. Additionally it efficiently maps Smalltalk method activation context to stack frames. The VM uses a JIT that maps bytecodes to native instructions and applies basic register allocation to reduce stack load. This situation is not a direct requirement for Benzo but it is assumed as given and thus not further discussed in detail.

However Benzo requires certain features that were not supported in the existing VM implementation. Mainly we need to generate executable code at runtime and run it. This requirement is essential and applies to any VM that wants to support dynamic code execution at runtime.

3.1.1 Executable Memory

We use standard Smalltalk objects to hold the generated native code. However, by default the object memory is not executable. This leaves two choices: mark the whole object memory executable or only move the objects with the native code to a special executable memory region. We took the path of least resistance and marked the whole object memory as executable. The other solution requires substantial changes for memory management. As the VM has a moving GC we only access high-level Smalltalk objects via an indirection from low-level code.

Another approach would have been to harness the fixed sized executable region used by the existing JIT. However the JIT space does not hold normal Smalltalk objects but special low-level structures and uses its own special GC.

3.1.2 VM Interaction

The standard way in Smalltalk to execute low-level code is to use a tag in the method definition. The following example shows such a method on the `Float` class.

```
* aNumber
    <primitive: 49>
    ↑ aNumber adaptToFloat: self andSend: #*
```

Here we use the primitive 49 to call a VM function which efficiently multiplies two floats. Figure 2-a describes the case where the primitive is successfully executed. However if the primitive is unable to do the operation, for instance if the argument `aNumber` is not a float, it will signal a failure which causes the VM to execute the fallback Smalltalk code in the method body. Fig. 2-b describes it.

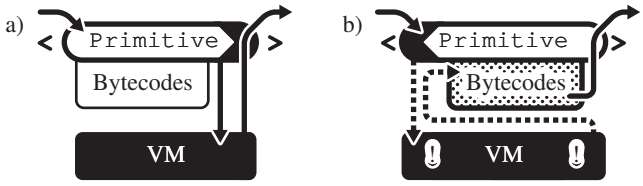

Figure 2: Generic primitive methods in Pharo: a) A primitive completely bypasses the bytecode, b) A failing primitive executes the Smalltalk bytecode as fallback.

Benzo uses the primitives as a gate to enter the low-level world from the language-side. The primitive then executes the native code generated and returns to language-side. The generated native code is appended inside the compiled method object. When the primitive is activated, it accesses the currently executed compiled method via a VM function. Figure 3 shows the structure of a Smalltalk compiled method that has native code attached to it. We see the primitive tag on top, followed by the literal frame which holds references to symbols and classes used in the method. The subsequent Smalltalk bytecode is the fallback code executed only if the primitive fails. Only then appears the native instructions. A marker at the end of the compiled method called trailer type is used to flag methods that actually have native code attached to them.

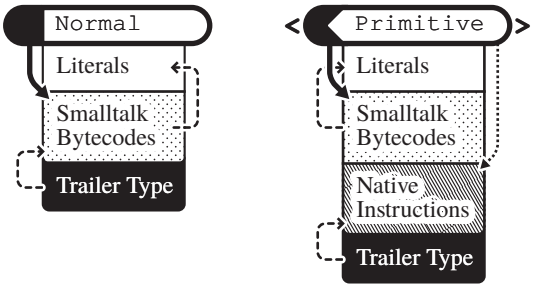

Figure 3: A standard Smalltalk compiled method on the left and a method with appended native instructions generated by Benzo.

Since compiled methods are first-class objects it is possible to modify them at runtime and append the native code. The primitive `primitiveNativeCall`, which is implemented by Benzo, is the responsible of running the native instructions in a Smalltalk method. The code example `interrupt3` shows a very basic application of our infrastructure. In Section 3.2 we describe how Benzo uses Smalltalk code to generate the native instructions, specifically Section 3.2.1 will explain more detailed examples.

```
interrupt3
    <primitive: 'primitiveNativeCall'
     module: 'BenzoPlugin' >
    Benzo generate: [ :asm | asm int3 ]
```

Listing 1: Smalltalk method using Benzo for low-level debugging.

3.1.3 Native Code Platform Interaction

To ensure that the code is compatible with the current platform a VM specific marker is expected at the beginning

of the native code on the compiled method. Upon activation Benzo compares this marker with the one from the current VM. If they don't match, Benzo signals a failure that causes the VM to evaluate the fallback Smalltalk code. With this elegant approach Benzo regenerates native code lazily on new platforms. Moreover, it does not have to flush the native code when the application is restarted on the same platform.

3.1.4 Garbage Collector Interaction

Compiled methods in Pharo have a special section, the literal frame, which stores objects referenced in the bytecodes. Bytecodes then only have indirect access to these objects by indexing into the literal frame. This simplifies the implementation of the garbage collector as it only has to scan the beginning of each method for possible references to objects. So the GC only tracks Smalltalk objects when they are in the method's literal frame.

The moving GC of the VM used for Pharo has a significant impact on the low-level code we can generate using Benzo. For instance it is not possible to statically refer to language-side objects from native code as object addresses changes after each garbage collection. Modifying the GC to support regions of non-moving objects would solve this problem. However we chose to minimize the number of low-level VM modification necessary to run our experiments and opted for a simpler solution. Like the existing compiled methods, Benzo's accesses language-side objects through an indirection.

For indirectly accessing objects the Pharo VM already features a special structure, named external roots. This array has a fixed-location in memory which can be used to access moving language-side objects. The GC updates the addresses in this VM structure after each run. Hence we have the static address of the external roots object as an entry point to statically access a Smalltalk objects.

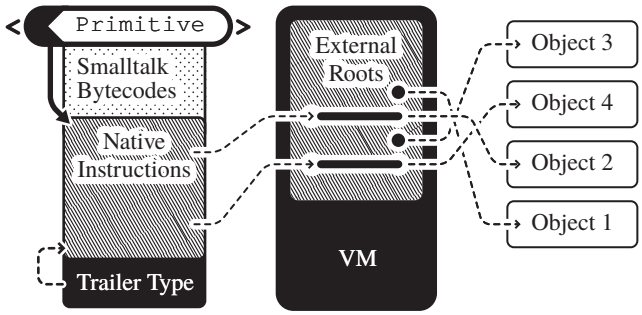

Figure 4: Pointers to objects registered as external roots are pinpointed at fixed offset in global VM-level object.

So for accessing Smalltalk objects within native code we first register it as an external root object and access it only indirectly. This means that for native code, instead of a method-local literal array we share a global literal array as shown in Figure 4. Benzo only adds an `Array` to the external root objects which is managed from language-side and administers all references.

3.1.5 JIT Interaction

When the Pharo VM starts the execution of dynamic generated code the execution environment changes slightly. Similarly, when entering primitives or plugin code that mode is left and a normal C level execution environment is reestablished until the primitive finishes and the VM jumps back to the jitted code. To avoid this context changes that imply a considerable performance overhead, we extend the VM to support inlining of native code in the JIT phase following the same strategy as other existing primitives which are inlined at JIT-level.

The Benzo prologue and epilogue used for managing the low-level stack are replaced by an adapted version for the JIT. The performance boost of this optimization is further discussed in Section 5.2.

3.1.6 Error Handling

Benzo provides an error handling facility that allows to return high-level error messages from the low-level code. The native code builder provides a helper method called `failWithMessage:` that generates the proper assembler instructions to return a full error message. This allows plugins to return clear and meaningful error codes, improving the debugging tasks and enabling a better interaction with users.

3.2 Benzo's Language-Side Implementation

We keep the interface to the low-level world minimal. The following describes the salient features in the high-level language-side of Benzo.

3.2.1 Code Generation

Benzo delegates native code generation to a full assembler written in Smalltalk. The following example shows how to use the assembler to generate the native code for moving `1` into the 32-bit register `EAX`.

```
ASM x86 generate: [ :asm |
    asm mov: 1 asUImm to: asm EAX ].
```

The implementation first creates a slightly more abstract intermediate format. The abstract operations can be extended by custom operations that may expand to several native instructions. For pragmatic reasons current implementation only supports `x86` and `x86-64`.

The plan is to improve the platform independence by implementing a more abstract domain specific language for Benzo low-level instructions.

The full runtime features of Pharo are available when generating native code. Hence complex instruction sequences can easily be delegated to other objects. In the following example we use a VM helper to instantiate an array, note that these are all standard Pharo message sends:

```
ASM x86 generate: [ :asm :helper | | register |
    register ← helper classArray.
    register ← helper
        instantiateClass: register
        indexableSize: 10
    asm mov: register to: asm resultRegister.
    ].
```

In this case the `#instantiateClass:indexableSize:` will generate the proper native code to call to a VM function and make sure that the side-effects of a possible GC run are handled properly. By default the value in the result register is returned back to the image, on `x86` this defaults to `EAX`. The VM helper exposes a basic, low-level interface to access objects and its properties. Additional methods cover the access of external roots described in Section 3.1.4. Section 4 will give more complete applications which are based on Benzo.

3.2.2 Code Activation

Benzo primitive is responsible for the native code activation which consists of three main steps:

- Check if there is native code in the actual compiled method and if it is compatible with the current platform.
- Generate native code if necessary.
- Activate the native code for execution.

The example in Listing 1 uses Benzo's generator to create and install the native code which would trigger a low-level interrupt. Behind the scenes Benzo adds some more information to the code as the already mentioned platform marker. For activation Benzo uses reflective features to restart the method containing the native code. Upon the second activation, after already generating the native code, Benzo moves the native code to the end of the compiled method and activates it. This mechanism is shown in Figure 5.

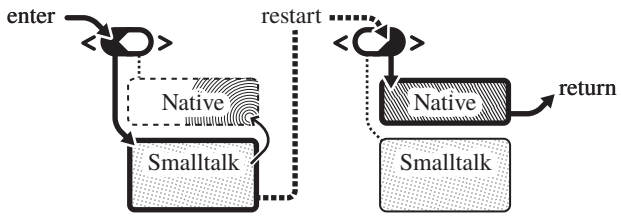

Figure 5: Native code activation with Benzo: The first call triggers the code generation. Then the method is restarted and the native code executed.

4. Benzo in Practice

In the following Section we will present a dynamic language-side implementation, based on Benzo, for each of the three examples mentioned in the introduction for extending language runtimes.

4.1 NativeBoost: a Benzo-based Foreign Function Interface

FFIs enable a programmer to call external functions without the need to implement additional VM extensions. NativeBoost [11] is a full FFI developed on top of Benzo. An FFI implementation consists of two main parts:

- **Execution:** calling external functions.
- **Marshalling:** converting data between the languages.

Typically most of these two parts are implemented in the VM with statically defined bindings to convert basic types such as integers, strings and floats between the different representations. Furthermore they provide entry points to find external functions by name in a certain external library. Relying on Benzo capability to dynamically generate and execute native code we developed a complete FFI at language-side. This way the VM no longer requires to have a specific FFI extension.

FFI at Language-side. The fact that via FFI we can call external functions makes it a perfect option to replace VM extensions defined at low-level side since FFI relies only on one generic low-level extension: the language-side has to be able to generate and subsequently call native instructions. A VM with a well-defined plugin infrastructure enforces the same level of separation. However unlike plugins, FFI bindings are implemented without crossing a language barrier. Most code for FFI bindings can be written at language-side in already existing familiar infrastructure. Furthermore, compared to a low-level plugin, a language-side library is easier to evolve and maintain.

FFI-based language extensions also provide a certain level of portability. Often the only artifact that has to be ported is the FFI plugin for the VM. In the optimal case the high-level FFI code is completely compatible. If the platform does not provide the same signature for the function, only the language-side code requires changes. This is preferable since the language-side part of the FFI code relies on better abstractions and infrastructure for debugging.

NativeBoost does not even depend on a specific VM plugin but on the generic infrastructure provided by Benzo. All the FFI is implemented at high-level language-side. Figure 6 shows how only the last step in calling an external function relies on low-level VM interaction. Section 4.1.2 explains the execution component details. Via reflection techniques NativeBoost provides a simple yet powerful marshalling library which is further described in Section 4.1.3.

4.1.1 NativeBoost in a Nutshell

A very simple example to illustrate the functionality of NativeBoost is to access the current environment variables. We do this by calling the `getenv` and `setenv` C functions. `getenv` takes a name as single argument and returns the value of that environment variable as a string.

```
getenv: name
    ↑ FFI call: 'String getenv(String name)'
```

In this example NativeBoost automatically detects that the arguments for the Smalltalk method are the same as for the low-level C function. The most important aspect about this example is that it is written with standard Smalltalk code. In figure 5 we show how NativeBoost lazily generates native code on the first method activation.

4.1.2 External Functions and Symbols

For a complete and practical FFI implementation the gathering of external function addresses is an imperative requirement. NativeBoost supports this for every platform. For instance, on UNIX-like systems NativeBoost achieves this by wrapping around the existing functions `dlopen`, used for opening shared libraries, and `dlsym`, used for returning function name addresses.

4.1.3 NativeBoost Symbiosis with Pharo

NativeBoost uses reflection capabilities to detect and marshall Smalltalk method arguments to C-level function arguments taking advantage of the full power of Smalltalk to support complex type conversions. This allows to have simpler declaration of FFI calls.

Argument Detection. NativeBoost automatically detects the arguments for the C function from the name given in its declaration. For instance, in the example of Section 4.1.1 the argument for `getenv` is found by looking at the method source code. In more complex setups the arguments of the method might not correspond to the order of the C function's arguments and a binding by name does the job.

Type Marshalling. NativeBoost automatically converts primitive types between the C world and Smalltalk. In the same previous example of `getenv` we replaced the `char *`

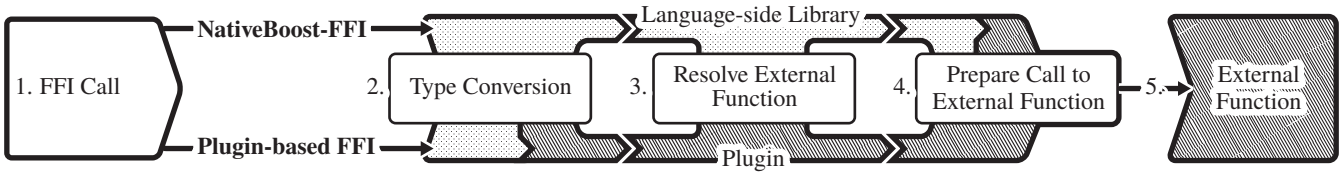

Figure 6: NativeBoost Overview: Unlike typical FFI implementations NativeBoost only resorts to the VM-level when actually calling the external function in step 4. Typical implementations already cross the low-level barrier during the type conversions at step 2.

from the original function signature with the single type `String`. This allows NativeBoost to automatically marshal the Smalltalk String into the corresponding C representation. From a Smalltalk point of view the original declaration `char *` is ambiguous. Smalltalk distinguishes between arrays of characters and a real string. For more elaborate type conversions such as C-level structs NativeBoost uses marshalling objects that reify the low-level common structures.

4.1.4 NativeBoost Performance

Compared to a static plugin-based FFI implementation NativeBoost has only a one-time startup overhead with its numbers shown in Section 5.2. Generating the native code at language-side is substantially slower than directly setting up all the conversions and calling the external functions from C code. In some cases the penalty for some compilation effort on NativeBoost is as high as a factor of 100 compared to classic approaches. Under the assumption that the method is called several times this overhead may be considered negligible. The following table shows a performance comparison of three different FFI implementations for Pharo Smalltalk.

	Call Time	Relative Time
NativeBoost	`10.53` ± `0.35` ms	1.0×
Alien	`31.09` ± `0.94` ms	≈ 3.0×
FFI	`19.55` ± `0.64` ms	≈ 1.9×

Table 1: Different FFI implementations in Pharo running `abs` with a single argument. Alien does marshalling at language-side while FFI does everything in C.

Table 1 measures the accumulative time of 100'000 FFI calls. Included in these numbers is at least one additional Smalltalk message send to activate the NativeBoost method containing the actual call to the C function. Each benchmark itself is run 1000 times and the average and standard deviation is taken. We also measured calls with more complex type conversions where the performance boost against Alien pronounced even more because NativeBoost's language-side marshalling is nativized. The JIT interaction described in Section 3.1.5 is also an important optimization factor especially when calling out small helper routines where the context switch from jitted mode is not negligible.

4.2 Reflective Primitives

Pharo VM is developed in a language that is a subset of Smalltalk known as Slang, which is transformed to C and then compiled using a standard C compiler. Slang basically has the same syntax as Smalltalk but is semantically constrained to expressions that can be resolved statically at compilation or code generation time and are compatible with C. Hence Slang's semantics are closer to C than to Smalltalk. The primitives of the language are written in Slang since are part of the VM.

That's why even in highly reflective languages like Smalltalk where almost every aspect of the language is available for inspection or modification [13] primitives can not be changed at runtime. Waterfall [2] is a JIT compiler that takes the standard primitive definitions from the code written in Slang and translates them to native code. This replaces the indirection via C that is used in the default compilation process for primitives. But given that Slang source code can be modified at runtime as any other Smalltalk method, Waterfall fosters primitives to be dynamically changed.

4.2.1 Waterfall Compiler Summary

From a high-level point of view the services provided by Waterfall can be outlined in two main functionalities:

- Compile Slang code on demand (lazily), at runtime and from language-side.
- Provide a clear interface for executing, also at runtime and from language-side, the native code generated by the compiler.

The first item allows to change the code of primitives at language-side and generate the corresponding native code when needed. Also it provides the potential to write methods or functionalities with the same Smalltalk syntax but with a static semantic. It consists essentially of a transformation toolchain that uses the AST that is generated by the standard Pharo compiler harnessing that Slang and Smalltalk have the same syntax. Then this AST representation is translated to native code enforcing C-like Slang semantics. The current prototype has only three fully implemented stages: Slang to AST, AST to an IR (between TAC and SSA) and finally AST or IR to native. The design is open for future additions at any level. One typical enhancement missing is having different levels of intermediate representations with various techniques on code optimization and register allocation strategies as modern compilers propose [5, Ch. 1].

The second item from above list is the responsible of presenting a clear interface that allows executing the dynamically generated native code. This includes for instance the gathering of positions of the VM internal symbols. Waterfall relies on NativeBoost, the Benzo-based FFI presented in the previous section, for interfacing with C libraries (`dlsym`). It also includes the linking between the two worlds: Smalltalk and native. Benzo is heavily used for providing this second main functionality.

Primitives in Smalltalk. As already partially explained, whenever a method is compiled with the `primitive` pragma

as shown in Section 3.1.2 a flag is set on the `CompiledMethod`. If the VM finds that the flag is set, it gets the number of the primitive and instead of interpreting the bytecodes it calls the corresponding function at VM-level[17]. The binding between primitives and numbers is described in a table indexed by number.

Smalltalk distinguishes two types of primitives: essential and non-essential primitives. Essential primitives are required for the bootstrapping and the essential operations of the language, such as creating a new object or activating a block. The second category of primitives are mainly used for optimization purposes.

Dynamically Interchangeable Primitives. Waterfall uses Benzo's mechanism for replacing primitive methods with customized nativized versions that are created dynamically as described in Section 3. This loophole of the language exploited by Waterfall provides the advantage of having the possibility to dynamically modifiy some VM behavior with a considerable much lower penalty on performance.

4.2.2 Benefits and Contribution

We identified two main benefits of changing VM primitives at runtime:

- Reducing VM complexity by implementing non-essential primitives reflectively at language-side.
- Dynamic instrumentation of primitives.

Reducing VM Complexity. In Section 2 we concluded that VM extensions are only justified in the presence of strong performance requirements. All non-essential primitives fall into that category. Using Waterfall these primitives may be implemented at language-side. This means that these primitives become first-class citizens of the high-level environment and thus evolve with less effort.

Instrumentation of Primitives. Essential primitives can not be fully implemented at language-side using Waterfall. These primitives are required for system startup. Hence they would trigger an endless recursion when booting up the system. However nothing prevents from replacing essential primitives at runtime with customized versions. We use Waterfall with primitives for efficient instrumentation purposes.

Actually it is absolutely possible to do instrumentation completely at language-side for non-essential primitives without Waterfall by accepting the performance penalty, but for essential primitives doing it is a very fragile task. The chances of accidentally invoking the same primitive in the language-side instrumentation code are high. Without very careful design the instrumentation code will thus trigger an endless recursion. Also performance issues could be prohibitive for language-side solutions. With Waterfall we can avoid these issues since the instrumentation code eventually will be implemented at the lowest level.

4.2.3 Performance Analysis

For comparing performance we implement a very simple integer operation primitive (>) using three different approaches. The first approach is the implementation with Waterfall. The second is to run the language-side implementation that is triggered whenever the standard primitive failed. Finally the fast standard primitive provided by the VM. We run the three approaches by measuring the cumulative time over one million primitive activations averaged over 100 runs. The absolute numbers are less important than the relative factor between them. We present the results of this experiment in Table 2.

	Running Time	Relative Time
VM	`6.4` ± `0.14` ms	1.0×
Waterfall	`22.8` ± `0.17` ms	≈ 3.6×
Reflective	`195.0` ± `0.16` ms	≈ 30.0×

Table 2: Comparing running time of different implementations of integer arithmetic primitive.

As expected Waterfall's solution outperforms pure reflective one by factor 9 to 10. Waterfall clearly outperforms a purely reflective solution since all the meta programming overhead for the intercession mechanism is avoided. This results thus makes a whole new set of runtime extensions feasible that were previously limited by their strong performance penalty. Furthermore the performance penalty over a completely optimized VM solution that has extreme optimization techniques, such as inlining and register allocation, is less than a factor of 4. Applying standard optimization techniques, not yet implemented in Waterfall, will almost sure improve these numbers even more.

4.3 Nabujito JIT Compiler

In this section we present Nabujito, a Benzo-based approach for a language-side JIT compiler. Nabujito goes even further than Waterfall using almost the same techniques. However instead of focusing on primitives, Nabujito generates native executable code for standard Smalltalk methods. Primitives tend to be more low-level, whereas Nabujito focuses on high-level Smalltalk code.

4.3.1 The JIT of the Pharo VM

The Pharo VM already comes with a JIT that translates bytecodes to native instructions. It transforms Smalltalk methods into slightly optimized native code at runtime. The main speed improvement comes from avoiding bytecode dispatching and by inlining certain known operations and primitives [6].

The most complex logic of the JIT infrastructure deals with the dynamic nature of the Smalltalk environment. `Methods` and `classes` can be changed at runtime and that has to be addressed by the JIT infrastructure. The JIT compiler, by which we refer in this context to the transformation of bytecodes to native code, represents a small part of the whole infrastructure. There exists more important stages as an additional register allocation pass to reduce the number of stack operations [26, 27]. The existing JIT infrastructure is implemented in Slang [8, Ch. 5] as the rest of the VM.

4.3.2 Limitations of Standard JIT Compilers

Since the JIT compiler itself is quite decoupled from the rest of the JIT infrastructure we believe that a hard-coded static and low-level implementation is not optimal for several reasons:

- Optimizing Smalltalk code requires strong interactions with the dynamic environment.
- Accessing language-side properties from the VM-side is hard.
- Changing the JIT compiler requires changes to the VM code.

- The JIT reimplements primitives for optimization reasons resulting in code duplication.

Optimizations Limits for Smalltalk. In Smalltalk methods tend to be very small and it is considered good practice to delegate behavior to other objects. That implies that several common optimization techniques for static languages do not work. The dynamic method activation do not provide enough context for a static compiler to optimize methods. Hence after inline caches and register allocation the next optimization technique is inlining. However inlining in a dynamic context is difficult and requires hooks at VM level to invalidate native code when the language-side changes. Since in Smalltalk compiling a method is handled completely with language-side code most of the infrastructure to get notified about method changes is already present.

Primitives in the Existing JIT. The existing JIT reimplements the most used primitives at VM-level. This is necessary for instance to guarantee fast integer operations. A typical example is the integer addition which has to deal with overflow checks and conversion of tagged integers. In Section 4.2 we describe how Waterfall suffers a similar requirement. Hence Waterfall manually defines such primitives in terms of native assembler instructions through the language-side Benzo interface. Nabujito, a language-side JIT compiler described on next section, reuses the same optimized primitives so we rely on a single optimized definition which is shared amongst all native code libraries.

4.3.3 Implementing Nabujito

Nabujito is an experimental JIT implementation which replaces the bytecode to native code translation of the existing JIT infrastructure with a dynamic language-side implementation. Nabujito is implemented mainly with a visitor strategy over the intermediate bytecode representation. Additionally we reimplemented using Benzo vital native routines for the JIT which are not directly exported by the VM.

Nabujito relies on the following VM-level infrastructure to manage and run native code:

- Fixed native code memory segments.
- Routines for switching contexts.
- Native stack management.

Dynamic Code Generation. To simplify the implementation we decide to manually trigger JIT compilation. For primitives known by Waterfall we rely on that infrastructure to generate the native code. For standard methods Nabujito takes the bytecodes and transforms them to native code.

It also applies optimizations such as creating low-level branches for Smalltalk level branching operations like `ifTrue:`. Optimizations for additional methods are all implemented flexibly at language-side. Wherever possible we reimplement the same behavior as the existing native JIT compiler.

Eventually the native code is ready and Benzo attaches it to the existing compiled method. When the language-side jitted code is activated Benzo ensures that we do not have to leave the JIT execution mode, and thus we can call methods at the same speed as the existing JIT. The benchmarks of section 4.3.4 show the empirical results.

4.3.4 Nabujito Performance

Performance is of course the main contribution of a JIT and it is imperative to analyze the efficiency of a language-side implementation.

Nabujito essentially generates the same native code as the VM counterpart. For the experiment we reimplement the C routines found in the VM JIT at language-side. There is no speed difference in the generated native code. However Nabujito is slower during the warm-up phase. Compilation of the native instructions will take considerably more time compared to the C implementation of the same bytecode to assembler transformation. However this is not critical for long-term applications.

	Compilation Time
Pharo Compiler	`71` $\pm$ `1` ms
Nabujito	`73` $\pm$ `1` ms

Table 3: Compilation efforts of the standard Smalltalk compiler in Pharo and Nabujito for the a simple method returning the constant `nil`.

In Table 3 we compare the compilation speed of the standard Pharo compiler and Nabujito. We measure the accumulated time spent to compile the method 1000 times. The average and deviation are taken over 100 runs. The Pharo compiler takes source code as input and outputs Smalltalk bytecodes. Nabujito takes bytecodes as input and outputs native code.

We see that in the simple case displayed in Table 3 Nabujito's compilation speed lies within the same range as the standard Smalltalk compiler. We expect that in the future we apply more low-level optimizations and thus increase the compilation time of Nabujito. However we have shown in the performance evaluation for NativeBoost, the Benzo-based FFI, in Section 4.1.4 that even a rather high one-time overhead is quickly amortized. Furthermore with Smalltalk's image approach the generated native code is persistent over several sessions. A subsequent restart of the same runtime will not cause the JIT to nativize the same methods it did during the last launch. Hence our approach is even valid for short-timed script-like applications as most of the methods will already be available in optimized native code from a previous run.

4.3.5 Outlook

One major performance optimization missing in both, the original VM-level JIT and Nabujito, is inlining. By inlining we are able to create methods that are potentially big enough for optimizations. However inlining is a difficult task in a highly dynamic language such as Smalltalk. Efficient inlining can only be performed with sufficient knowledge of the system. Accessing this high-level information from within the VM is cumbersome and requires duplication of language-side reflective features. We are convinced that with Nabujito we simplify this task significantly. The JIT lives on the same level as the information it needs relying on the already present reflective features of Smalltalk.

5. Implementation Issues

The aspects concerning security for this kind of low-level capabilities over high-level languages allow for much discussion and controversy. Performance is the other most discussed is-

sue in the system programming domain and we exposed the results of Benzo related to it.

5.1 Security in Reflective Low-level Programming

Benzo breaks the security aspects provided in high-level languages such as memory safety or proper exception handling [23]. However the implications are not different from any other FFI implementation used in high-level languages. Direct use of low-level native instructions poses a security risk to the system. There has been detailed research in how to make FFI implementations more secure. Typically the compiler statically ensures that no compromising structures leave the VM-realm [18]. By analyzing the internal usage pattern of the external function it is possible to further reduce the risk of accidentally modifying vital internal VM structures. By shielding of the VM internal structures from the external world we effectively limit the risk but at the price of limiting also the power of an FFI. We show in Section 2 why FFI existent solutions are not powerful enough for certain types of extensions that are important for us. Security risks are one of the reasons exposed for this limitations.

Besides of the inherent security problems of FFI there is the whole reflective power of the Smalltalk environment as a security risk. Smalltalk allows a programmer to change classes and methods at runtime. There exist even methods that dynamically replace all references to an object with another one in the entire system. For many other high-level languages such functionality is not accessible from language-side or not present at all in the runtime. Some Smalltalk language features, such as the live instance migration [17], rely on this reflective capabilities and are vital for the developer experience. Hence we can conclude that Benzo poses the same security risks as other essential architectural decisions that the Smalltalk environment promotes.

We believe that security has to be addressed in a more general way at language-side and not restricting the possibilities of the developers. If we can enforce proper security constraints at language-side it is possible to encapsulate dangerous behavior in a controlled domain. Only with such a solution are we able to provide security in a Smalltalk-like environment.

5.2 Performance

Benzo allows the generation of efficient native code. We already showed that the generated native-code from language-side only causes a one-time overhead on its initial creation. Thereafter it is cached for later activations. We also argued that for the three Benzo proof of concepts examples proposed in Section 4 this overhead can be neglected. Furthermore, for the FFI implementation we show in Table 1 how we outperform the existing FFI implementations due to more specific native code. The performance gain by the execution of custom-made native code outweighs the one-time cost of language-side code generation.

Benzo's close interaction with the JIT described in Section 3.1.5 further reduces the reoccurring costs of calling native-code.

LuaJIT follows the same approach for their FFI library [1].

Our conclusion is that even a high one-time compilation overhead has little influence on the overall performance of the system. Hence the benefits of reflective low-level programming outweigh.

6. Related Work

QUICKTALK [7] follows a similar approach as Waterfall. However Ballard et al. focus mostly on the development of a complex compiler for a new Smalltalk dialect. Using type annotations QUICKTALK allows for statically typing methods. By inlining methods and eliminating the bytecode dispatch overhead by generating native code QUICKTALK outperforms interpreted bytecode methods. Compared to Waterfall QUICKTALK does not allow to leave the language-side environment and interact closely with the VM. Hence it is not possible to use QUICKTALK to modify essential primitives.

High-level low-level programming [16] encourage to use high-level languages for system programming. Frampton et al. present a low-level framework packaged as `org.vmmagic`, which is used as system interface for Jikes, an experimental Java VM. Additionally their framework is successfully used in MMTK [9] which is used independently in several other projects. The `org.vmmagic` package is much more elaborate than Benzo but it is tailored towards Java with static types. Methods have to be annotated to use low-level functionality. Additionally the strong separation between low-level code and runtime does not allow for reflective extensions of the runtime. Finally, they do not support the execution and not even generation of custom assembly code in the fly.

Other related approaches are VM generation frameworks in general. They try to abstract away the complexity of the VM and use high-level languages as compiler infrastructure. A very successful research project is Jikes Research VM [20]. It uses Java to metacircularly define a Java runtime which then generates the final VM. A similar framework is PyPy [29] a VM framework including an efficient JIT. PyPy uses a restricted subset of the Python language named RPython which is then translated to various low-level backends such as C or LLVM code. There exist several different high-level language VM implementations on top of PyPy such as Smalltalk [10] or Prolog. However its main focus lies on an efficient Python interpreter.

Other high-level languages such as Lua leverage FFI performance by using a close interaction with the JIT. LuaJIT [1] for instance is an efficient Lua implementation that inlines FFI calls directly into the JIT compiled code. Similar to Benzo this allows to minimize the constant overhead by generating custom-made native code. The LuaJIT runtime is mainly written in C which has clearly different semantics than Lua itself. Compared to our approach the efficient VM implementation suffers from the shortcomings described in Section 2.3.

Kell and Irwin [21] take a different look at interacting with external libraries. They advocate a Python VM that allows for dynamically shared objects with external libraries. It uses the low-level DWARF debugging information present in the external libraries to gather enough metadata to automatically generate FFIs. However they do not focus on the reflective interaction with low-level code and the resulting benefits.

7. Conclusions

We presented Benzo a reflective low-level programming framework written in a dynamic high-level language. Benzo is an integral approach for reflective high-level low-level programming. Using Benzo we efficiently implemented at language-side three distinct language feature extensions that typically reside at VM level.

Benzo promotes interaction with the low-level world by dynamically generating native code from language-side. This allows to exploit the underlying platform capabilities only when strongly needed without leaving the development platform and through a high-level programming interface. Benzo advocates the use of development tools and abstraction level of the high-level language for as much as possible or desired.

With high-level reflection capabilities combined with efficient low-level code we manage to do dynamic primitive instrumentation and reuse the code for primitive operations which is duplicated on the standard JIT approach. We also show that since Benzo caches native code transparently at language-side our JIT compiler poses only a one-time overhead when generating native code. Our mature FFI implementation outperforms an existing C-FFI implementation by a factor 1.5 even though we control every aspect from the language-side.

Benzo shows that promoting clear interfaces for controlling low-level code completely from language-side produces efficient solutions for system programming requirements without resorting to pure low-level solutions. We showed that combining the abstraction provided by high-level languages with the complete and precise powerful system programming capabilities of low-level languages is not only possible but profitable. Furthermore we manage to considerably reduce complexity and code duplication which results in better maintainability.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011-IRSES. Also we would like to thank Marcus Denker, Damien Pollet and Ciprian Teodorov for kindly reviewing earlier drafts of our paper.

References

[1] LuaJIT FFI Library. `http://luajit.org/ext_ffi.html`.

[2] Waterfall. `http://lafhis.dc.uba.ar/waterfall`.

[3] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 314–324, New York, NY, USA, 1999. ACM.

[4] A. Andersen. A note on reflection in Python 1.5. In *Lancaster University*, 1998.

[5] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998.

[6] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[7] M. B. Ballard, D. Maier, and A. W. Brock. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986.

[8] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[9] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[10] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week – implementing a Smalltalk VM in PyPy. *Self-Sustaining Systems*, pages 123–139, 2008.

[11] C. Bruni, L. Fabresse, S. Ducasse, and I. Stasenko. Language-side foreign function interfaces with nativeboost. In *Submitted to International Workshop on Smalltalk Technologies 2013*, 2013.

[12] G. Cooper. DTrace: Dynamic tracing in Oracle Solaris, Mac OS X, and free BSD by Brendan Gregg and Jim Mauro. *SIGSOFT Softw. Eng. Notes*, 37(1):34, Jan. 2012.

[13] M. Denker, J. Ressia, O. Greevy, and O. Nierstrasz. Modeling features at runtime. In *Proceedings of MODELS 2010 Part II*, volume 6395 of *LNCS*, pages 138–152. Springer-Verlag, Oct. 2010.

[14] K. Fisher, R. Pucella, and J. Reppy. Data-level interoperability. In *Electronic Notes in Theoretical Computer Science*, page 2001, 2000.

[15] D. Flanagan and Y. Matsumoto. *The Ruby programming language*. O'Reilly Media, Incorporated, 2008.

[16] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, Eliot, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[17] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[18] M. Hirzel and R. Grimm. Jeannie: granting Java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 19–38, New York, NY, USA, 2007. ACM.

[19] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *OOPSLA'97: Proceedings of the 12th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–326. ACM Press, Nov. 1997.

[20] The Jikes research virtual machine. `http://jikesrvm.sourceforge.net/`.

[21] S. Kell and C. Irwin. Virtual machines should be invisible. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 6. ACM, 2011.

[22] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM.

[24] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, Dec. 1987.

[25] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. Proceedings of the Reflection '96 Conference, 1996.

[26] E. Miranda. Context management in VisualWorks 5i, 1999.

[27] E. Miranda. The Cog Smalltalk virtual machine. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*. ACM, 2011.

[28] J. Reppy and C. Song. Application-specific foreign-interface generation. In *Proceedings of the 5th international conference on Generative programming and component engineer-*

ing, GPCE '06, pages 49–58, New York, NY, USA, 2006. ACM.

[29] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[30] D. Ungar and R. B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA, 2007. ACM.

[31] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 11–20, New York, NY, USA, 2005. ACM.

[32] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45:59–72, Oct. 2010.

[33] J. Vraný, J. Kurš, and C. Gittinger. Efficient method lookup customization for Smalltalk. In *Proceedings of the 50th international conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 124–139, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.