

Towards Self-aware Virtual Machines

Vers des machines virtuelles auto-décrites

THÈSE

présentée et soutenue publiquement le 16. May 2014

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Camillo Bruni

Composition du jury


Président : Christophe DONY
Rapporteur : Christophe DONY, Laurence TRATT, Gaël THOMAS
Directeur de thèse : Stéphane DUCASSE (Directeur de recherche – INRIA Lille Nord-Europe)
Co-Encadreur de thèse : Marcus DENKER

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 41414

Copyright © 2014 by Camillo Bruni

RMoD
Inria Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley
59650 Villeneuve d’Ascq
France
<http://rmod.inria.fr/>

 This work is licensed under a *Creative Commons Attribution-ShareAlike 4.0 International License*.

Acknowledgments

I would like to thank my thesis supervisors Stéphane Ducasse and Marcus Denker for allowing me to do a Ph.D at the RMoD group.

I thank the thesis reviewers and jury members Christophe Dony, Gaël Thomas and Laurence Tratt for kindly reviewing my thesis and providing me valuable feedback.

I would like to express my gratitude to Igor Stasenko for providing `BENZO` and `NATIVEBOOST`, and Guido Chari for allowing me to use `WATERFALL` for validation purposes.

I thank Camille Teruel for helping me translate the abstract of this thesis.

For remarks on earlier versions of this thesis I thank Stefan Marr and Damien Pollet.

Abstract

High-level languages implement reflection which allows a language runtime to inspect and alter its own execution and state. These high-level languages typically run on top of virtual machines (vms) which have been built to create an abstraction layer over hardware. Due to the isolating nature of the vm, reflection is generally limited to the language-side. Several research vms overcome this separation and provide a unified model where there is no more a clear distinction between language-side and vm-side. In such a language runtime it is possible to reflectively modify vm components from language-side as they reside on the same abstraction layer.

In this dissertation we follow the same global direction towards a unified language-runtime or self-aware vm. However, instead of looking for a holistic solution we focus on a minimal approach. Instead of using a custom tailored language runtime we use dynamic native code activation from language-side on top of an existing vm.

We first present BENZO our framework for dynamic native code activation. BENZO provides a generic but low-level interface to the vm internals.

Based on this framework we then evaluate several applications that typically require direct vm support. We show first how BENZO is used to build an efficient FFI interface, allowing for a more structured access to vm internal functions. To evaluate the limitations of BENZO we target two more applications: dynamic primitives and a language-side JIT compiler. Both of them require a tight interaction with the underlying vm

Keywords:. Virtual machine, high-level low-level programming, high-level language, dynamic native code generation.

Résumé

Les langages de haut-niveau supportent des opérations réflexives qui permettent à l' environnement d'exécution d'un langage d'inspecter et de changer son propre état et sa propre exécution. Ces langages de haut-niveau s'exécutent normalement sur une machine virtuelle (VM) qui ajoute une couche d'abstraction au-dessus du matériel. À cause de cette séparation, peu d'opérations réflexives sont disponibles pour inspecter et modifier la VM. Plusieurs VMs expérimentales offrent de telles opérations réflexives en proposant un modèle unifié qui ne distingue pas la couche VM de la couche langage.

Dans cette thèse, nous suivons une approche similaire qui propose un environnement d'exécution unifié et auto-décrit. Nous nous intéressons à une solution minimale. Au lieu de dépendre de modifications d'une VM, nous générons dynamiquement du code natif depuis la couche langage.

Nous présentons BENZO, un framework pour la génération dynamique de code natif. BENZO fournit une interface générique et de bas-niveau pour accéder aux fonctionnalités fondamentales de la VM.

Grâce à BENZO, nous analysons plusieurs applications qui nécessitent un accès direct à la VM. Nous montrons comment BENZO peut être utilisé pour implémenter une librairie de Foreign Function Interfaces, permettant de faciliter l'accès aux fonctionnalités bas-niveau de la VM. Pour évaluer les limitations de BENZO, nous visons deux autres applications: la génération dynamique de primitive et un compilateur JIT (Just-In-Time). Ces deux applications doivent changer le comportement de la VM. Pour cela, elles ont besoin d'une interaction poussée avec la VM.

Mot clés: Machine virtuelle, langage de haut-niveau, génération dynamique de code natif

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Contributions	3
1.3	Artifacts	3
1.4	Outline	4
2	Background	7
2.1	Reflection	8
2.1.1	Scoping Reflection: Partial Reflection	9
2.1.2	Reflection in vms and Language Runtimes	10
2.2	High-level Languages and vms	14
2.2.1	Metacircular vms	14
2.2.2	Compile-time Reified vms	17
2.2.3	Runtime Reified or Self-aware vms	20
2.3	High-level Low-Level Applications	23
2.3.1	Foreign Function Interfaces (FFI)	24
2.3.2	Just-in-time Compilation	24
2.4	Problem 1: Dynamic High-level Low-level Programming	25
2.5	Problem 2: Extending Reflection by High-level Low-Level Programming	26
2.6	Problem 3: Minimal Interface for High-level Low-level Programming	26
2.7	Summary and Outlook	26
3	BENZO: Low-level Glue in PHARO	29
3.1	Background	29
3.1.1	Requirements	30
3.1.2	BENZO a Framework for Reflective High-level Low-level Programming	30
3.1.3	BENZO Applications	31
3.2	The BENZO Framework	31
3.2.1	VM Context	32
3.2.2	BENZO's Language-Side Implementation	37
3.3	BENZO in Practice	39
3.3.1	NATIVEBOOST: BENZO-based Foreign Function Interface	39
3.3.2	Reflective Primitives	40
3.3.3	NABUJITO JIT Compiler Prototype Outline	41
3.4	Performance	43
3.5	Extending the Language Runtime	44
3.5.1	Language-side Library	45
3.5.2	Language-side Reflective Extensions	46
3.5.3	VM Extensions / Plugins	47
3.5.4	Hybrid Extensions	49
3.6	Limitations of Benzo	50
3.6.1	Robustness	50

3.6.2	Low-level Debugging	52
3.6.3	Platform Independence	53
3.7	Conclusion and Summary	54
4	Validation: FFI	57
4.1	Background	58
4.2	NATIVEBOOST-FFI: an Introduction	60
4.2.1	Simple Callout	60
4.2.2	Callout with Parameters	61
4.2.3	Automatic Marshalling of Known Types	61
4.2.4	Supporting new types	63
4.2.5	Callbacks	65
4.2.6	Overview of NATIVEBOOST-FFI Internals	67
4.3	Performance Evaluation	68
4.3.1	Callout Overhead	69
4.3.2	Marshalling Overhead for Primitive Types	70
4.3.3	Using Complex Structures	71
4.3.4	Callbacks	72
4.4	Implementation Details	72
4.4.1	Generating Native Code	72
4.4.2	Activating Native Code	74
4.5	NATIVEBOOST Limitations	74
4.5.1	Difficult Debug Cycles	74
4.5.2	Platform Independence	75
4.5.3	Limited Expressiveness	76
4.5.4	Startup Recursion	77
4.5.5	Performance	79
4.6	Conclusion	79
5	BENZO Prototype Application Validation	81
5.1	WATERFALL: Dynamic Primitives	82
5.1.1	Background	82
5.1.2	WATERFALL'S Contribution	84
5.1.3	WATERFALL Implementation	86
5.1.4	WATERFALL Validation	88
5.1.5	WATERFALL Limitations and Outlook	91
5.2	NABUJITO: Language-side JIT Prototype	93
5.2.1	Background	94
5.2.2	NABUJITO Implementation	98
5.2.3	NABUJITO Validation	102
5.2.4	NABUJITO Limitations and Future Work	103
5.3	BENZO Applications: Outlook and Summary	106
6	Future Work	107
6.1	Background	108
6.1.1	PINOCCHIO VM	108
6.2	Language-side Improvements	109
6.2.1	Improved Domain Specific Inspectors	109
6.2.2	Virtual CPU an Assembler DSL	110
6.2.3	Barrier-free Low-level Interaction	114

6.3	vm-level Improvements	118
6.3.1	Missing vm-level Reification	118
6.4	Summary	119
7	Conclusion	121
7.1	Contributions	122
7.2	Published Papers	122
7.3	Software Artifacts	124
7.4	Impact of the Thesis	125
	Bibliography	127
A	Appendix	133
A.1	PHARO Programming Language	133
A.1.1	Minimal Syntax	134
A.1.2	Message Sending	134
A.1.3	Precedence	135
A.1.4	Cascading Messages	136
A.1.5	Blocks	136
A.1.6	Methods	137

INTRODUCTION

Chapter

Contents

1.1	Problem Statement	2
1.2	Contributions	3
1.3	Artifacts	3
1.4	Outline	4

Several high-level languages support dynamic reflection, the capability of a language to reason about itself. In an extended form, reflection allows a program to alter its own structures and execution at runtime. Many high-level programming languages run on top of a virtual machine (vm) which provides certain advantages from running directly on the underlying hardware. Many high-level language vms pursue a strict separation between language-side and vm-side. vms for instance provide automatic memory management or use platform agnostic instructions such as bytecodes. These properties allow a programming language to develop independently from the underlying hardware. Originally vms are built in performance oriented low-level programming languages such as C which on their own support little or no reflection at runtime. Hence, rather incidentally, reflection is limited to the language-side.

However, there are other vms that are implemented using high-level languages which support reflection themselves. More specifically we see that metacircular vms encourage advanced reflective features and new ways of interacting with the low-level vm world. Metacircular vms are implemented in the same language they support. Typically this enables a more flexible building process where more high-level structures survive the compilation process. The final language-runtime can profit from this and support *high-level low-level programming* [25]. This term was coined by the use of JAVA to describe low-level components in the memory management toolkit of the JIKES vm. In several research vms this concept is used to implement typically isolated vm components at language-side. But instead of generating a native version at vm generation time, these components are evaluated at language-side, with the only difference that they have the capability to directly interact with low-level code. This means that the original separation of language-side and vm is no longer evident.

A language-runtime where vm components are implemented at language-side also enables extended forms of reflection. With the same reflective tools it is now possible to inspect and alter vm-level objects. However, this ap-

proach requires substantial effort as the VM has to be designed from ground up in a new way. In this thesis we would like to follow a different approach with the same goals in mind. Instead of a holistic approach, we want to identify a minimal interface for performing high-level low-level programming or even extend reflection down to the VM. We look for an approach which works on top of an existing high-level language VM.

To answer these questions we propose the high-level low-level programming framework **BENZO** written for **PHARO**. In the core **BENZO** allows us to dynamically activate native code from language-side. This adds a primitive yet generic interface to the low-level VM world. To validate **BENZO** we describe three distinct applications built on top of it.

FFI: The first one is an efficient foreign function interface (**FFI**) library that is built at language-side without additional VM support. Our **FFI** library outperforms existing solutions on **PHARO**.

Dynamic Primitives: The second applications uses **BENZO** to dynamically generate and modify **PHARO** primitives by reusing the metacircular VM sources. By combining high-level reflection and **BENZO**'s low-level performance we outperform pure **PHARO**-based primitive instrumentation.

Language-side JIT: As a third, prototype application we show how **BENZO** is used to build a language-side **JIT**. Our prototype shows the limits of possible VM interactions using the **BENZO** framework.

1.1 Problem Statement

Following the problem description listed in the above introduction we identified the following abstract concerns with existing reflective languages and their VMs.

- Reflection and in special behavioral reflection comes at a significant cost due to reification overhead.
- Intercession is limited to language-side. VMs are not accessible from language-side and they are usually have no reflective properties at runtime.
- Existing approaches to a unified model between the VM and the language-side are holistic, there is no intermediate solution available.

Out of these general problems concerning reflection in high-level languages we see that they have a low-level root. To address the unification of the language-side and VM-side we have to grant more access to the language-side. This includes interacting directly with low-level native instructions.

A similar problem has been solved by applying high-level low-level programming in a more static environment [1, 25]. The approach outlined by Frampton et al. uses a high-level framework to generate native code at compile-time. We see that their approach has not yet been applied in a more dynamic environment where native code has to be generated at runtime. Hence we focus on the following concrete problems we wish to solve in this thesis.

Problem 1: High-level low-level programming is not available at runtime and from language-side.

Problem 2: Intercession is limited to language-side. vms are not accessible from language-side and they are usually have no reflective properties at runtime.

Problem 3: High-level low-level programming has not yet been provided as an incremental extension to an existing language runtime.

1.2 Contributions

To support high-level low-level programming in a dynamic context we identify that native code generation at language-side is essential. Hence we validate this concept by implementing a language-side framework for native code generation and execution with minimal vm changes. Using this framework we identify the limitation of such an approach by evaluating several typical vm-level applications.

1.3 Artifacts

We present now our contributions of this dissertation addressing the previously identified problems concerning high-level low-level programming in a dynamic language:

BENZO is a high-level low-level programming framework written in PHARO¹.

The core functionality of BENZO is to dynamically execute native-code generated at language-side. Our framework requires minimal changes to an existing vm and three custom primitives to support dynamic code activation, the majority of BENZO is implemented as accessible language-side code. BENZO allows us to directly communicate with the low-level world and thus hoist typical vm-level applications to the language-side.

¹<http://pharo.org/>

NATIVEBOOST is a **BENZO**-based foreign function interface (FFI). **NATIVEBOOST** generates customized native code at language-side, both being flexible and efficient at the same time. **NATIVEBOOST** outperforms other existing FFI solutions on the **PHARO** platform, making it an ideal evaluation for the **BENZO** framework.

NABUJITO is a prototype JIT compiler based on **BENZO**. **NABUJITO** generates the same native code as the VM-level JIT by compiling the high-level byte-code intermediate format at language-side. Our **BENZO**-based JIT prototype reuses existing VM-level infrastructure and focuses only on the dynamic code generation. However, since there is no well-defined interface with the VM **NABUJITO** requires an extended VM with an improved JIT interface to dynamically install native code.

ASMJIT is an assembler framework written in **PHARO**. **ASMJIT** is the low-level backend for the previously mentioned **BENZO** framework. We extended the existing assembler framework to support the full 64-bit x86 instruction set.

1.4 Outline

Chapter 2 sheds light on the context of this work. We present a quick overview of language-side reflection followed by a development of VM-level reflection. We find that mostly metacircular VMs provide limited VM-level reflection and thus we present several high-level language VMs falling into this category. We conclude that there is only two research VM that has a uniform model for VM and language-side. Among them is **PINOCCHIO** a research **SMALLTALK** VM we contributed to previous to working on this dissertation.

Chapter 3 describes a high-level low-level programming framework named **BENZO**. The core functionality of **BENZO** is to dynamically execute native-code generated at language-side. **BENZO** allows us to hoist typical VM plugins to the language-side. Furthermore we show how code caching makes **BENZO** efficient and users essentially only pay a one-time overhead for generating the native code.

Chapter 4 presents **NATIVEBOOST**, a stable foreign function interface (FFI) implementation that is entirely written at language-side using **BENZO**. **NATIVEBOOST** is a real-world validation of **BENZO** as it combines both language-side flexibility with VM-level performance. We show in detail how **NATIVEBOOST** outperforms other existing FFI solutions on **PHARO**.

Chapter 5 focuses on two further **BENZO** applications. In the first part we present **WATERFALL** a framework for dynamically generating primitives

at runtime. `WATERFALL` extends the concept of metacircularity to the running language by reusing the same sources for dynamic primitives that were previously used to generate the static `VM` artifact. In a first validation we show how `WATERFALL` outperforms other reflective language-side solutions to instrument primitives.

In a second part of Chapter 5 we present `NABUJITO` a prototype `JIT` compiler that is based on `BENZO`. `NABUJITO` shows the limitations of the `BENZO` approach as it required a customized `VM` to communicate with the existing `JIT` interface for native code. Our prototype implementation generates the same native code as the existing `VM`-level `JIT`, however, it is currently limited to simple expressions. `NABUJITO` shows that for certain applications a well-define interface with the low-level components of the `VM` is required.

Chapter 6 summarizes the limitations of `BENZO` and its application. Furthermore we list undergoing efforts on the `BENZO` infrastructure and future work.

Chapter 7 concludes the dissertation.

BACKGROUND

Contents

2.1	Reflection	8
2.2	High-level Languages and vms	14
2.3	High-level Low-Level Applications	23
2.4	Problem 1: Dynamic High-level Low-level Programming	25
2.5	Problem 2: Extending Reflection by High-level Low-Level Programming	26
2.6	Problem 3: Minimal Interface for High-level Low-level Programming	26
2.7	Summary and Outlook	26

Introduction

In this chapter we present the related work to this dissertation. We first present a quick overview of language-side reflection followed by a description how reflection developed for vms.

In the context of this thesis we are mainly interested in behavioral reflection that requires strong support from the underlying vm. We identify that this form of reflection is rather costly, namely due to its late binding that does not allow for static optimizations. Following to this, we present how different techniques of partial behavior reflection are used to limit the cost of reflection. We see that reflection can be more efficient with more vm support available. At this point we outline the evolution of reflection in high-level languages with an ultimate goal being a language that has full control over its own vm and thus blurring the line between language-side and vm-side.

The second part of this background chapter focuses on different kinds of vms and how they are built. We find that metacircular vms provide a good match to our idea of unified language runtime. After presenting several recent metacircular vm projects we conclude that most of them limit reflection at the vm-level to compile time. Only a couple of research vms have a uniform model that spans across all abstraction levels. Among them is PINOCCHIO a research SMALLTALK vm we contributed to previous to working on this dissertation.

This chapter finishes by presenting a detailed description of the problem statement and a final outlook of the upcoming chapters in the light of the found problems.

2.1 Reflection

In this section we give a quick overview of the core features of reflection. A system is said to be reflective if it is capable to reason about itself. Typically we distinguish two forms of reflective access: structural and behavioral [34]. Structural reflection is concerned with the static structure of a program, while behavioral reflection focuses on the dynamic part of a running program. Orthogonally to the previous categorization we distinguish between introspection and intercession. For introspection we only access a reified concept, whereas for intercession we alter the reified representation.

Structural Reflection means to access the static structure of a program. A typical example¹ is to access the class of an object at runtime.

```
'a string' class.
```

An example of structural intercession is to reflectively modify an instance variable of an object.

```
aCar instVarNamed: #driver put: Person new.
```

Behavioral Reflection means to directly interact with the running program. For instance this includes reflectively activating a method.

```
#Dictionary asClass perform: #new
```

Another more complex example to dynamically switch the execution context and resend the current method with another receiver.

```
thisContext restartWithNewReceiver: Object new
```

Accessing the receiver of the current method through the execution context is an example of behavioral introspection.

```
thisContext receiver.
```

There is not always a clear separation between the two types of reflection possible. For instance it is possible to add new methods which requires structural reflection. At the same we alter the future program execution which also implies that the action was behavior reflection. Typically we see that behavioral reflection stops at the granularity of a method. For instance in PHARO by default it is not possible to directly alter execution on a sub-method level [22].

Additional to separating reflection upon the representation it accesses, we distinguish what actions are performed on the reified representations. Both of the following properties can apply for structural and behavioral reflection.

¹Throughout this dissertation we use PHARO code examples. The syntax and the basic semantics are explained in Section A.1

Introspection is the form of reflection that does not alter the reified representation. An example of this is the previous code excerpt where we access the class of an object.

Intercession implies that the underlying representation is altered. Going back to the previous example that would for instance mean to change the class of an existing object.

```
MyClass adoptInstance: anObject.
```

These four categories summarize the general properties of reflection. In the following text we use the term "*reflection*" or "*dynamic reflection*" as a short form of unanticipated behavioral *and* structural reflection both for performing intercession and introspection. In the course of this dissertation we will introduce an additional category to distinguish between reflection that happens at language-side accessible by the developer and reflection that happens inside the VM.

From now on we use a circular arrow (Ω) to symbolize dynamic reflection in a figure.

2.1.1 Scoping Reflection: Partial Reflection

Reflection brings great power to a programming language. However, especially behavioral reflection is linked to a significant overhead. For instance the previous example of the reified execution context in SMALLTALK requires restricts the optimizations at VM-level. And more general, most reification comes at great costs [35]. Hence already from a performance point of view it is natural to limit the scope of reflective behavior. For instance, using wrapped methods to alter execution has a wide-spread effect on the system. Thus, there is also a motivation to limit the effect on evaluation introduced by reflection. We will now discuss several axes along which we can limit the use of reflection.

Time: Of course the most obvious axis is time itself. Behavioral reflection implies that the reflective properties are accessed or modified dynamically. This implies that the use of reflection changes over the course of evaluation. By dynamically adding or removing the reflective code we have time-delimited reflection.

Type: Another natural delimiter for reflection is the type of an object. In a typical object-oriented system this is the common case. Methods are implemented on different classes which themselves define the type of their instances. Custom methods that are added to a class alter the behavior of all its instances.

Reference: Starting from the concept of a proxy object we find another possibility to limit reflection by reference. Arnaud et al. describe a modified PHARO runtime where the concept of a reference is fully reified as a so called handles [6]. Handles allow programmers to install new behavior and even state on a single reference, without influencing the rest of the system.

Context: The effects of reflection can also be limited by the dynamic execution context [44]. An example of that is the concept of tower of interpreters. During the development of PINOCCHIO an intermediate version of the SMALLTALK interpreter featured this special execution scheme [51]. It allows the programmer to switch the current interpreter. This way an expression is evaluated with altered semantics. The solution presented in PINOCCHIO does not globally replace the interpreter but only for the given expression. Hence once the expression returns, the modifications and the implied overhead are gone.

Tanter et. al. describe REFLEX [45] a partial behavioral reflection system on top of JAVA. We see similar limitation mechanisms for the applications of aspects [32], which resembles intercession. However, typically the systems using aspects have to be prepared statically upfront with little means to change them at runtime. Aspects can be used to globally modify a system and introduced code snippets in defined points, for instance before each method invocation. Though they share an interesting concepts of limiting the introduced overhead using a pointcuts. These are conditionals that are dynamically before evaluating aspects.

Both REFLEX and aspects require the underlying system to be prepared upfront. Unanticipated behavioral reflection is not directly possible. Typically it is only possible to enable or disable the reflective features that have been prepared upfront. Röthlisberger et al. propose GEPETTO a system [42] that enables true unanticipated behavioral reflection on top of SMALLTALK. GEPETTO provides a high-level API to install behavior reflection.

2.1.2 Reflection in vms and Language Runtimes

So far we have given a basic introduction to the different types of reflection and how the effects of reflection can be limited by certain properties. We omitted how reflection is provided in the first place in a language runtime. For instance, simple cases involve giving access to the class of an object or the possibility to reflectively invoke a method at runtime. In very reflective languages the VM provides access to the current execution context for introspection and even modification. The latter one has a significant influence on the underlying VM architecture preventing certain low-level optimization which

would shadow the access to certain context information. What we see is that the meta-level enables reflection but usually is not reflective by its own.

Following the principle that everything is an object one might assume that this also includes the VM as it is already highly involved in supporting reflection. Typically the VM is implemented in C or C++ which have no reflection. However, the JIT is common exception as it has to interact dynamically with the language-side. For instance the JIT has to be aware of classes and the methods within. In dynamic and reflective system the JIT has to be made aware of language-side changes to properly update or invalidate the generated native code. What we see is that the JIT accesses structural information from the language-side. However, the language-side is not capable of accessing VM-level information. The closest VM interaction point typically is the bytecode generated at language-side and handed over to the VM for execution. Yet, this provides only a crude one-way interaction [29]. Certain VMs provide debugging or inspection interfaces which are used by external tools to access or modify the VM internals. Even though technically the same VM debugging interface can be used by the language itself it is not common.

For highly reflective and dynamic languages we see a certain mismatch. On the one hand, it is possible to virtually change and modify everything at language-side. On the other hand, it is generally not possible to reflectively alter the VM from language side. The underlying VM tends to ensure security by isolation for instance by using a defined bytecode set as execution base. However, the border to the VM can be crossed in several ways. For instance, not all operations are implemented with safety in mind but performance, and thus might expose the VM. Another edge-case is the use of external native libraries which are used for performance critical functionality and tend to be run without the same protection as bytecode-based code.

Thus we see that the boundary to the VM can be crossed in several ways. Proper separation is possible for a static language which restricts the reflective power or prohibits the use of custom external libraries. However, this is clearly not the case for contemporary dynamic programming languages. Since arbitrary code changes are possible at language-side the VM-level boundary for reflection seems incidental. Furthermore, research suggests that this separation is a limiting factor when focusing on low-level interaction with external functionality [30]. Reflection down to the VM level is possible if we leave certain security and performance concerns aside.

To further analyze this hypothesis we need to track the evolution of dynamic reflection in programming languages. In our analysis we omit languages that do not run on top of a VM. Typically low-level or system programming languages fall into this category which implies unrestricted operations. For instance C or even assembler is used to generate self-modifying

programs. Another example might be C++ which supports compile-time reflection, but again does not run on top of a VM. We are interested in language runtimes that are built around a VM which introduces the aforementioned separation of high-level language-side and low-level VM.

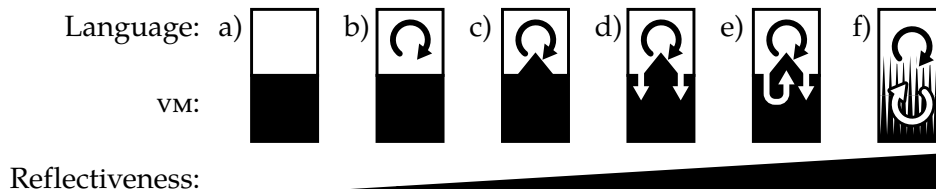


Figure 2.1: Evolution of reflection in high-level language runtimes starting from a non-reflective language in a) and ending in a self-aware system f) where reflection transcends the VM-language barrier. Circular arrows denote the use of reflection.

- a) **Language-side without reflection:** A language in this category requires a VM to run but has no reflective properties. This includes early-stage languages such as the original Pascal-P system [39]. This is rather an exception, since typically languages without reflection also lack the underlying VM and are compiled to native code. Even with portability in mind it is possible to use for instance C as intermediate language which compiles under most platforms.
- b) **Language-side with limited reflection:** The next step in the evolution of reflection is a language runtime with a VM that support only certain static reflection. This might include structural reflection whose required information can be prepared upfront during the compilation phase. Such a system has no support for unanticipated reflection as there is no support from the VM to dynamically reify concepts. A VM with a JIT in this category can perform strong optimizations and take full advantage of the runtime information.
- c) **Language-side extended reflection:** The third category of high-level language runtimes has extended reflection with strong support from the underlying VM. We put PHARO, SMALLTALK implementations or SELF in this category of languages. The VM supports complex reification of otherwise non-accessible concepts. These language runtime support extended behavioral reflection, for instance accessing and modifying the execution context. The supported reflective capabilities can not be anticipated, and thus require strong support from the underlying VM. At this stage the VM-level optimizations are a balance between restricting the supported language or sacrificing speed.

- d) **Language-side introspection of the VM:** The VM support for reflection is highly extended compared to the previous category. Instead of a hidden property, certain VM-level concepts are made explicitly accessible to the running language. Up to some extent this is similar to language-side structural reflection as the VM only supports only a restricted interface which is defined at compile-time. In this category the language can only read (introspect) VM properties. This might include reading out JIT related properties such as performance counters or type annotations. Typically these operations are supported by VMs running in debug mode, which enables remote introspection. However, this does not imply that the language-runtime is capable of doing so reflectively.
- e) **Limited language-side intercession of the VM:** The previous category allows the language-side to safely read VM-level properties. If we follow the same path as the language-side evolution of reflection the next step is to allow for modification at VM-level. Such a language-runtime has a dynamic interface to change certain properties of the VM. However, the VM is still not fully reflective in the sense that not all VM concepts are reified. This essentially limits the language-side to simple interactions and changes to the VM itself. At this point the VM can no longer guarantee safety by isolating the language-side from all the low-level details.
- f) **Self-aware VM:** We classify in the last category dynamic language-runtimes that have no longer a clear separation of VM and language-side. The same reflective properties equally apply to language-side and the VM. The way to achieve this is by flattening out the intermediate VM and let the language-side directly control everything. Currently there are several research VMs which can be classified as self-aware VMs: The PINOCCHIO VM [50] is partially self-aware but in control of the underlying execution and the KLEIN VM is fully reflective [48]. Unsurprisingly we find that these VMs are built metacircularly, they are written in a subset or the same language they support. In the following Section 2.2 we will discuss in more details metacircular VMs.

From this overview of the evolution of reflection in high-level languages and their VMs we see that there certain language runtimes that provide a form of VM-level reflection. Which is a clear indicator that the clear separation between language-side and VM-side is mostly incidental. However, we see that there is only little research about self-aware VMs or reflective VMs. According to our overview, only a few research language runtimes would classify as self-aware. Combined with the previous two categories d and e, we see that metacircular VMs encourage extended reflection. In the following Section 2.2 we are now going to describe in more detail how metacircular VMs are built

and what their contribution to the high-level low-level interaction is.

2.2 High-level Languages and vms

High-level language vms are inherent complex pieces of software. They have to combine two rather extreme goals: abstraction and performance. We have seen that the required abstraction for the running high-level language has a strong influence on the vm design. At the same time the hard performance requirement requires precise interaction with the underlying hardware. This goes even so far that specialized hardware is conceived to match the performance requirements [20, 36, 43, 46].

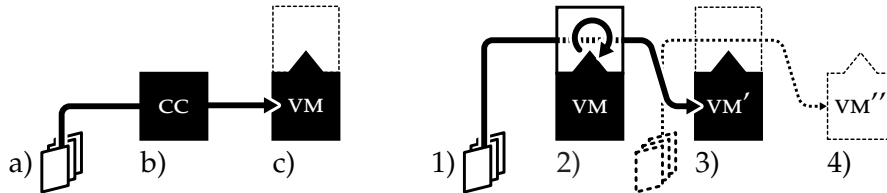
The early vms focused on interpreting an abstract instruction set (bytecodes). The benefits are twofold. On the one hand the bytecodes guarantee certain platform independence by abstracting away from the cpu specific instruction set. On the other hand bytecodes allow to encode complex operations into little space both serving the hard memory constraints of the hardware and simplifying the design of a compiler. Obviously this abstraction gain comes at a cost and ever since the first vms were built research and industry strive to reduce the interpretation overhead. An efficient way to improve performance is to use a just in time compiler (jit) that dynamically generates native code from the bytecode [23]. In this case the bytecode becomes an intermediate representation (ir) for a bigger compiler infrastructure. However, jit compilers are notoriously complex as they crosscut many vm components. At the same time they crosscut all abstraction layers; they have to access high-level information from the running bytecodes and manage native code at the same time. Similar complexity applies to the automatic memory management present in most high-level language vms. Garbage Collectors (gc) evolved from simple helpers to complex software artifacts that for instance support concurrent garbage collection [20].

The increased complexity of the vms lead to more novel approaches on how to build vms. vms are still build for a big part in C or C++ for performance reasons. However, there are more high-level approaches that try to simplify creating vms by using building blocks [26]. In the following sections we are shedding light on metacircular vms which are programmed in the same language they in the end support.

2.2.1 Metacircular vms

The ever growing complexity of vms and the abstraction mismatch between the vm definition language and the final interpreted language lead to a new movement that tried to reduce complexity. Among the vms using higher-level languages or frameworks to reduce the development effort the metacircular

building process stands out. Unlike the classical vm which is built in C and compiled to the a binary, a metacircular vm is written in the same language that it provides in the end. The following figure highlights the most evident differences between a classical and a metacircular approach.



Classical vm Compilation

- a) vm sources typically written in C or C++
- b) Compilation of the vm sources using a C or C++ compiler
- c) Final Binary

Metacircular vm Compilation

1. vm sources written in a high-level language, the same as the final vm supports
2. Compilation of the vm sources happens by evaluating the vm sources, allowing for compile-time reflection
3. New vm' binary built using an existing version of the vm
4. The new vm Binary can be used to compile again a new vm''

Using the same language for developing the vm has several advantages. Usually the vm is in great contrast to language-side libraries on the same platform. This is due to the low-level nature of the vm. Using a high-level language certain implementation details can be hidden. Furthermore the metacircular approach provides the vm developer with the same tools as a language-side programmer. Typically this leads to faster development.

Inside the metacircular vm community we see different approaches with varying levels of abstractions and reuse. When compared, we find differences in how metacircular vms build vm components (GC, JIT) and how the bootstrap or compilation of the new vm works. We see metacircular vms that use the high-level language as an advanced macro systems. In a sense an extended version of C++'s templates. Other approaches use the full reflective power of the high-level runtime to simplify code. And even more advanced system automatically provide the vm developer with GC or a JIT compiler. We will now elaborate in more detail how metacircular vms are constructed.

Language Property Synthesis. In the classical C-based vm approach all vm components have to be explicitly build. Each vm is a one of a kind with custom interpreter and a specialized memory manager. Using high-level vm frameworks it is possible to provide the vm developer with prefabricated compo-

nents. For instance it is possible to simply parametrize a pre-made GC to reduce development effort. Looking at the evolution of metacircular VMs we see both approaches. For instance pseudo metacircular solutions like the SQUEAK VM [28] work more like a high-level C macro system. The high-level language is used to generate C code which is then further compiled to the final VM binary. VM components are declared in a very explicit style, again not much different from C++. Memory for VM-level structures has to be managed in the same way as its C++ counterpart by explicitly allocating and freeing objects. On the other side we have VM frameworks like PYPY for the PYTHON language that provide automatic GC and JIT support. Here the developer writes a new VM in almost the same way as a normal PYTHON program. In the ideal case only certain hints are necessary to create a JIT.

Bootstrap Process. A crucial step during the development with the metacircular VMs is the bootstrap of the new VM. We distinguish mainly between two approaches, indirect bootstrap and direct bootstrap.

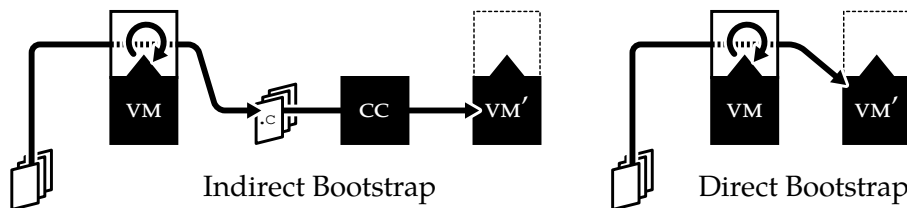


Figure 2.2: The indirect bootstrap on the left uses generated sources (.c) to compile a new VM' with a C compiler (CC). The direct bootstrap on the right directly creates a new binary without the use of an intermediate low-level language.

Indirect Bootstrap: Metacircular VMs with an indirect bootstrap use an intermediate language to compile a new VM binary. A typical example of this approach is SQUEAK and PYPY using C. Both of these systems imply a complete C compilation stack. The advantage of this approach is that C is heavily optimized thus reducing the development effort for the VM framework. However, C already hides a lot of low-level details away. Typically the VM framework has to work around these limitations when working directly with native code for instance in the JIT. We have explicitly seen these limitations while working on the PINOCCHIO VM.

Direct Bootstrap: Metacircular VMs with a direct bootstrap are directly in charge of generating the native code for the final binary. We have seen in PINOCCHIO that many C-level optimizations have only limited impact on the final speed. A major speedup is achieved by using a native stack and directly generating native code instead of using a bytecode interpreter.

Hence the VM will probably require an assembler framework which in return can be used for the direct bootstrap. This means that only limited additional efforts are necessary for a direct bootstrap. As a result the direct bootstrap allows full control of how the final binary will look like.

2.2.2 Compile-time Reified vms

After presented the technical background of metacircular vms we are presenting several concrete implementations in more detail. In this first part present vms that focus on compile-time reflection. In Section 2.2.3 we will then focus on a list of vms that reify their components and allow for a more close interaction with the language-side.

SQUEAK SMALLTALK VM

The SQUEAK VM [28] is of importance in the context of this work. Its core building system is still in active use for the COG VM² which extends SQUEAK with a JIT. The COG VM is used as default by the PHARO³ programming language. SQUEAK is built around a SMALLTALK dialect called SLANG that is exported to C to be compiled to the final VM binary. Additionally the SLANG sources can be interpreted to provide an interactive simulator of the VM, including full graphical support.

SLANG is limited to the functionality that can be expressed with standard C code. SLANG in this case is mostly a high-level C preprocessor. Even though SLANG basically has the same syntax as SMALLTALK it 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. Unlike later metacircular frameworks SQUEAK uses little or no compile-time reflection to simplify the VM designs. However, class composition help structuring the sources. Next to the SLANG source which account for the biggest part of the interpreter code some OS-related code and plugins are written in C. To facilitate the interaction with the pure C part SLANG supports inline C expressions and type annotations.

A great achievement of the SQUEAK VM is a simulator environment that enables programmers to interact dynamically with the running VM sources. The simulator is capable of running a complete SQUEAK SMALLTALK image including graphical user interface. This means that programmers can change the sources of the running VM and see the immediate effects in the simulator. The simulator itself works by setting up a byte array which servers as native

²<http://www.mirandabanda.org/cogblog/>

³<http://pharo.org/>

memory. Then the `vm` sources written in `SLANG` are interpreted by the `vm` of the development environment.

We see that `SQUEAK` is a pseudo metacircular `vm` that uses an indirect bootstrap process. The newly created `vm` does not absorb any features from the host environment. Yet according to long-time `vm` programmers the `SQUEAK` infrastructure is more productive than a comparable C++ or pure C project.

JIKES: High-level low-level Programming in with `MMTK`

`JIKES` (former `JALAPEÑO`) is an early metacircular research `vm` for `JAVA` [2]. The `JIKES` `vm` features several different garbage collectors and does not execute bytecodes but directly compiles to native code. With metacircularity in mind `JIKES` does not resort to a low-level programming language such as C for these typically low-level `vm` components. Instead they are written in `JAVA` as well using a high-level low-level programming framework.

The `JIKES` `vm` had performance as a major goal, hence direct unobstructed interaction with the low-level world is necessary using a specialized framework. High-level low-level programming [25] is mentioned the first time in the context of the `JIKES` `vm` project. The goal of high-level low-level programming is to provide high-level abstractions to simplify low-level programming. Essentially this is the same motivation that drives the metacircular `vm` community.

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 a separate project, the memory management toolkit (`MMTK`) [10] which is used independently in several other projects. The `org.vmmagic` package introduces highly controlled low-level interaction in a statically type context. In their framework, methods have to be annotated to enable the use of low-level functionality.

MAXINE `JAVA` `VM`

`MAXINE` is a metacircular `JAVA` `vm` [55] focused on an efficient developer experience. Typically `vm` frameworks focus on abstraction at the code-level which should yield simpler code and thus help reducing development efforts. However, in most situations the programmer is still forced to use existing unspecific tools for instance to debug the `vm`. In contrast to that, the `MAXINE` `vm` provides dedicated tools to interact with the `vm` in development. `MAXINE` uses abstract and high-level representations of `vm`-level concepts and consistently exposes them throughout the development process. Inspectors at multiple abstraction levels are readily available while debugging, giving insights to the complete `vm` state. `MAXINE` provides an excellent navigation for gener-

ated native code by providing links back to language-side objects as well as other native code and symbols.

Even though the MAXINE projects follows an approach where reflection is only used at compile-time, the development tools themselves provide a live interaction with the running VM artifact. However, the VM itself is not reflective as it is not directly built to reason about itself. This means that when debugging the VM it behaves almost like a live SMALLTALK image where a complete interaction with the underlying system is possible. We identify this as crucial, as most of the time is spent debugging, notably on inadequate tools like gdb due to lack of alternatives. Hence having a specific debuggers and inspectors greatly improve the interaction with the VM artifact.

PyPy Toolchain

PyPy⁴ is a PYTHON-based high-level VM framework [41]. PyPy's major focus lies on an efficient PYTHON interpreter. However, it has been successfully used to build vms for other languages including SMALLTALK [12]. Interpreters are written in a type-inferable subset of PYTHON called RPYTHON. The underlying PyPy infrastructure automatically provides memory management and JIT compilation. Instead of explicitly providing these features, a VM developer hints certain information to the PyPy framework to improve the generation of a GC or JIT.

PyPy follows a different approach from the previously presented VM generation frameworks. For instance, in SQUEAK and JIKES the final VM implementation is not much different from an implementation done directly in a low-level language. The programmer specifies all the components of the VM explicitly, either by implementing them directly or using a provided library. Compared to the more static C and C++ these VM generation frameworks make the compilation phase more tangible. SMALLTALK in SQUEAK or JAVA in JIKES or MAXINE fulfill the purpose of the template system in C++ or the restricted macro system in C. For the explicit implementation part PyPy is no different. However, certain features for the final VM are directly absorbed from the underlying PyPy infrastructure. For instance, the JIT support or the GC are not explicitly implemented but provided by the PyPy framework itself. This is a big difference to the other VM frameworks as it allows programmers to write the VM in a more high-level fashion. For instance in SQUEAK memory allocation, even for VM-level objects, has to be performed explicitly. Whereas in PyPy the garbage collection is left to the underlying VM building infrastructure. This approach allows RPYTHON vms to behave like standard PYTHON programs.

⁴<http://pypy.org/>

Much like the automatic memory management, PyPy provides a tracing JIT generator [11]. By default the VM programmer does not write an explicit JIT in PyPy. Instead the VM code is annotated to guide the underlying tracing JIT generator. This means at VM compilation time a specific tracing JIT is created for the given meta information. As a result, the JIT can track high-level loops in the final interpreted language. Again, this is similar to PyPy's GC, both are provided as a service and do not have to be programmed explicitly. Instead, the VM programmer tweaks parameters of the JIT or GC.

2.2.3 Runtime Reified or Self-aware VMs

The VMs presented so far have little or no self-awareness. VM generation frameworks allow a high amount of reflection at VM compile time. This meta information is typically compiled away. This is somewhat similar to what happens with templates in a C++-based VM. The VM frameworks themselves behave like a static language on their own. As a result, the final VM artifact has no access to the underlying definition anymore.

As an example we might have several VM components represented as high-level objects at compile or VM generation time. These objects have a class and methods attached, information that is reflectively accessible. However, once the VM is compiled down to native code, most of this information is lost. What is left is native code with low-level instructions that allow little or no reasoning about the original high-level structure.

We have shown in Section 2.1.2 how a potential evolution of reflection in a high-level language looks. We concluded that the evolution of language-side reflection implies a similar evolution at VM-level. More behavioral reflection at language-side requires more concepts to be reified in the VM itself. This requirement is conflicting with the previously described loss of reification at VM-level.

We are now going to present VMs that behave significantly different. Unlike the previous ones, they no longer make a clear distinction between the static VM and the dynamic language-side.

DWARFPYTHON

DWARFPYTHON [30] is a PYTHON implementation that aims at a barrier-free low-level interaction. It emerged from an earlier PARATHON which used DWARF debugging information from external libraries to facilitate foreign function interfaces. DWARFPYTHON takes this idea further. Additionally to describe low-level code, DWARFPYTHON uses the DWARF metamodel to describe PYTHON code and data. This is depicted in Figure 2.3. This approach has the advantage that the very same debugging mechanism applies for

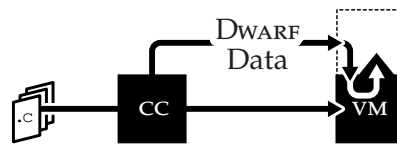


Figure 2.3: DWARFPYTHON reifies the low-level VM by using the DWARF Debugging at runtime. The DWARF information is generated by the default C compiler (CC) for C debuggers.

low-level code, for instance written in C, and for high-level PYTHON code. Thus DWARFPYTHON essentially unifies the previously decoupled VM with the language-side.

PINOCCHIO VM

PINOCCHIO [52] is a research SMALLTALK environment that directly uses native code instead of bytecodes. The only execution base is native code which is directly generated by the language-side compiler.

PINOCCHIO is built from a kernel derived originally from a PHARO image. For the bootstrap classes, objects and methods are exported into binary, native images and linked together with a standard C linker to a final executable. For simplicity we also rely on a very small part of C code to provide essential primitive, for instance used for file handling. Additionally we specified part of the bootstrap for the SMALLTALK object model in plain C code. However, besides that, all the other code is written and developed directly in SMALLTALK.

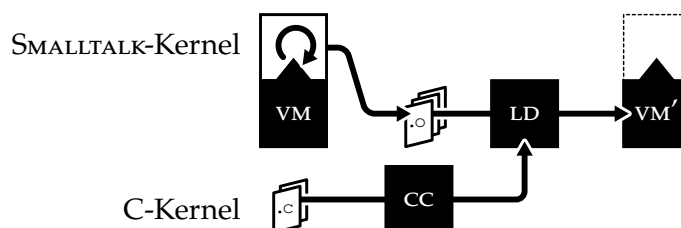


Figure 2.4: PINOCCHIO's Bootstrap directly generates binary images (.o) and combines them with a simple kernel compile from C sources using a standard C linker (LD).

An important aspect of PINOCCHIO is that the method lookup is expressed in terms of normal SMALLTALK code. Typically this code statically resides in the VM, thus at a different meta-level. Hence this implies for most systems that the lookup can not be modified without altering the VM itself. However, expressing the lookup in terms for normal language-side code introduces a recursive dependencies during the bootstrap. In order to run the lookup

code expressed in `SMALLTALK` code, we have to perform message sends. These, in return, require an already working lookup mechanism. Hence, without a taking special care, a language-side lookup method will lead to infinite recursion during startup. We resolved this problem in `PINOCCHIO` by directly interacting with the low-level execution format which among other things relies on inline caches to improve performance. The important property of inline caches is that they bypass the slow language-side lookup by directly jumping to the last activated method at a send-site. This is exactly the behavior we need to prevent recursion during the startup. Hence, when generating the native code for the bootstrap, we prefill all the inline caches of the methods required to perform a full method lookup. As a result, when running requiring the first real method lookup, the lookup code itself is running perfectly on the prefilled inline caches. What we achieve is a flexible connection between the low-level world and the high-level language-side. During execution the `VM` jumps freely between what previously was native `VM`-level code and interpretation of language-side code.

From an architectural point of view, `PINOCCHIO` is performing almost a direct bootstrap. Besides the small C kernel, the language-side code is directly compiled to native code. As a result, `PINOCCHIO` only requires a single compiler for native code, during bootstrap and at runtime. Hence, a separate `JIT` implementation is not required.

The most obvious shortcoming of `PINOCCHIO` is the lack of its own garbage collector. Instead of investing time into a separate well-defined GC `PINOCCHIO` relies on the conservative Boehm GC⁵ built for C programs. The Boehm GC is sufficiently fast to run `PINOCCHIO` as a prototype. `PINOCCHIO` lacks the necessary reification at level of the object layout to properly implement a GC. All the notion about the object layout in memory are hard-coded in the compiler in several places. Work was undertaken to put first-class object layouts in place and delegate memory allocation and field access to these meta objects. Yet, at the current state `PINOCCHIO` has not incorporated this in the compiler core.

`PINOCCHIO` is self-aware in the sense that it controls native code generation and lookup at a single abstraction level. There is no distinction between `VM`-level code and language-side code.

MIST a C-less `SMALLTALK` Implementation

`MIST`⁶ is another prototype `SMALLTALK` `VM` that follows similar goals as the `PINOCCHIO` `VM`. As well, it no longer uses a bytecode interpreter but only relies on native code. However, it goes one step further than `PINOCCHIO` by not relying on any C-based infrastructure. `MIST` implements its own linker to build

⁵http://www.hpl.hp.com/personal/Hans_Boehm/gc/

⁶<http://mist-project.org/>

the final executable. Hence unlike PINOCCHIO it does not require kernel primitives written in C. MIST brings its own implementation to directly perform system calls from within the language.

KLEIN VM

KLEIN⁷ is a metacircular VM for the SELF programming language that has no separation into VM and language [48]. KLEIN performs a direct bootstrap (see Figure 2.2) much like the aforementioned PINOCCHIO or MIST VM. Hence KLEIN does not use an intermediate low-level language to bootstrap the system.

It is important to point out that the reification of the VM-level survives the code generation or compilation time. Instead the VM structures are represented as real SELF objects. Hence the KLEIN VM supports true VM-level reflection since there is only a single code base.

Additionally to the advances in reflection and metacircularity, KLEIN focuses on fast compilation turnarounds to allow for a responsive development process. Which is unlike for instance the SQUEAK VM where a full VM bootstrap takes an order of minutes on modern hardware. KLEIN also supports advanced mirror-based debugging tools to inspect and modify a remote VM.

Development on the KLEIN VM stopped in 2009 and left the KLEIN VM in fairly usable state. Like PINOCCHIO it currently lacks a dedicated GC. Yet, it proved that it is possible and build a language-runtime without the classical separation of the language-side and the VM. From the literature presented about the KLEIN project we see a strong focus on the improvements of the development tools. The fact that the language-runtime allows VM-level reflection to change the VM dynamically is not directly mentioned in the literature. While we see the practical limitations of changing the VM at runtime we would like to open the doors to this new form of reflection.

2.3 High-level Low-Level Applications

In a high-level language and its VM we find several applications that span across multiple abstraction levels. We are discussing now two applications, Foreign Function Interfaces and the JIT compiler in more detail. Both explicitly require strong low-level interactions to perform their task. For instance, the MAXINE VM presented in Section 2.2.2 explicitly implements these applications using high-level low-level programming. This makes these two applications an interesting target for the evaluation of our dynamic approach to high-level low-level programming.

⁷<http://kleinvm.sourceforge.net/>

2.3.1 Foreign Function Interfaces (FFI)

Typical `SMALLTALK` systems are isolated from the low-level world and provide only limited interoperability with C libraries. However there are notable exceptions: `ÉTOILÉ` and `SMALLTALK/X`.

Chisnall presents the Pragmatic `SMALLTALK` Compiler [19], part of the `ÉTOILÉ` project, which focuses on close interaction with the C world. The main goal of this work is to reuse existing libraries and thus reduce duplicated effort. The author highlights the expressiveness of `SMALLTALK` to support this goal. In this `SMALLTALK` implementation multiple languages can be mixed efficiently. It is possible to mix Objective-C, `SMALLTALK` code. All these operations can be performed dynamically at runtime. Unlike our approach, `ÉTOILÉ` aims at a complete new style of runtime environment without a VM. Compared to that, `NATIVEBOOST` is a very lightweight solution.

Other dynamic high-level languages such as `LUA` leverage FFI performance by using a close interaction with the JIT. `LUAJIT`⁸ for instance is a very efficient `LUA` implementation with a tracing JIT. `LUAJIT` is interesting in terms of FFI implementations as it directly inlines FFI callouts into the JIT compiled code. Similar to `NATIVEBOOST` this allows one 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.

On a more abstract level, high-level low-level programming [25] encourage to use high-level languages for system programming. Frampton et al. present a low-level framework which is used as system interface for `JIKES`, an experimental `JAVA` VM. However their approach focuses on a static solution. 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 on the fly.

Kell and Irwin [30] 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.

2.3.2 Just-in-time Compilation

There is a vast amount of scientific literature when it comes to JIT optimizers. However, they focus on the optimization opportunities itself such as different compilation strategies or an efficient GC interaction. In the context of our work

⁸<https://github.com/jmckaskill/luaffi/>

compiler-based optimizations are of second importance since we focus on the hybrid nature of a system that interacts with the low-level VM world.

Jan Vraný et al. present a SMALLTALK with an explicit meta-object-protocol allowing for method lookup customization at language-side [53]. Their customized SMALLTALK/X VM has an extended lookup mechanism where each class can specialize the lookup with a user definable `LookupObject`. Hence for each message send the VM first checks if the receiver's class provides a `LookupObject`. By default this is not the case and the VM falls back to the standard hierarchical SMALLTALK method lookup which is hard-coded in the VM. However, if the receiver class returns a proper `LookupObject` the VM delegates the lookup to this user-defined object. The `LookupObject` is invoked with context information about the message send including access to the low-level lookup cache. While the other context information is important for new lookup schemes, the exposed cache provides an simplistic interface for the JIT. If the language-side lookup uses the provided cache it is still possible to implement efficient caching at VM-level.

A similar, albeit simpler approach, was provided in the research SMALLTALK VM PINOCCHIO [50]. There the message lookup is fully implemented at language-side, but unlike the SMALLTALK/X solution only the context information required for a standard SMALLTALK lookup is provided. More explicitly, PINOCCHIO does not provide access to an internal cache which could be used for speeding up more elaborate lookup customizations.

The two projects presented only implicitly deal with the JIT interaction. However, they provide evidence about high-level customizations for a part of the execution. In both projects it is possible to dynamically customize a static core VM concept. While it is possible to modify the lookup mechanism in many VM generation frameworks, this does not extend to the runtime

2.4 Problem 1: Dynamic High-level Low-level Programming

We have seen in the presented VMs that a tight integration with the low-level code is indispensable. Relying on an intermediate solution such as C with in-lined assembler expressions does not scale well. Typically it is troublesome to circumvent the aggressive optimizations applied by the C compiler in order to get the desired native code.

When working with metacircular VMs it is natural to implement a framework for maintaining the low-level code. Such high-level low-level programming [25] is used at compile time or VM generation time to create the necessary native code. However, we have seen that the same frameworks are not directly available in the final language-runtime. The JIT might make use

of such a framework at runtime, though that part is hidden in the VM itself. Hence we see an opportunity to use high-level low-level programming in a dynamic context and at language-side to implement new functionality.

2.5 Problem 2: Extending Reflection by High-level Low-Level Programming

In the case of the classical separation of the VM from the language-side, reflection stays isolated at language-side. However, this is different in a self-aware language-runtime supporting a reflective language. Due to the lack of clear separation between language-side and VM-side, the same or almost the same reflective properties should apply to both sides. Though this is a rather idealistic goal, as we have identified only few research VMs that have a unified model and thus are self-aware.

We have identified that for instance the KLEIN VM would be perfectly capable of performing reflection on components that typically belong to the VM. However, to our best knowledge we could not identify any publicly available work that leverages this fact.

2.6 Problem 3: Minimal Interface for High-level Low-level Programming

We have seen in this chapter that several research VMs provide a unified model where there is no longer a distinction between language-side and VM-side. In these systems VM components can be changed with the same reflective tools as language-side objects. These are whole language-runtime solutions, typically this is not a feature that can be added to an existing VM without big changes to the runtime.

In the context of this thesis we identify a minimal interface for high-level low-level programming. Instead of providing a new VM we want an incremental solution that extends the existing language-runtime.

2.7 Summary and Outlook

In this chapter we gave an overview of the related work of this thesis. We started by outlining the concepts of reflection and what implications it has on the performance. We have shown how partial behavioral reflection is used to limit the costs of reification. Out of this we have seen that for many reflective features specific VM support is required. Thus we presented in Section 2.1.2 a detailed description on how the evolution of reflection affects the VM. At the

end of the scale we describe a language-runtime that has no longer a clear distinction between language-side code and an isolated VM. In such a self-aware VM or unified language runtime, reflection equally affects the language-side and the VM.

In the second part of this chapter we discussed existing VM implementations. We focus on metacircular VMs as they are already a good match to a unified language runtime. They use reflection at compilation time to simplify the process of building a VM. However, even though reflection is widely used in these frameworks, it is restricted to the compile time. The final VM artifact has no reflective capabilities, it only provides the necessary interface to enable reflection at language-side. Hence, we present in a second group VMs that focus on a unified model.

We addressed the identified problems in the following way:

Chapter 3 describes a dynamic high-level low-level programming framework named *BENZO*. The core functionality of *BENZO* is to dynamically execute native-code generated at language-side.

Chapter 4 presents *NATIVEBOOST*, a stable foreign function interface (FFI) implementation that is entirely written at language-side using *BENZO*. *NATIVEBOOST* is a real-world validation of *BENZO* as it combines both language-side flexibility with VM-level performance.

Chapter 5 focuses on two further *BENZO* applications that extend the reflective capabilities of *PHARO* using high-level low-level programming. In the first part we present *WATERFALL* a framework for dynamically generating primitives at runtime. *WATERFALL* extends the concept of metacircularity to the running language by reusing the same sources for dynamic primitives that were previously used to generate the static VM artifact.

In a second part of Chapter 5 we present *NABUJITO* a prototype JIT compiler that is based on *BENZO*. *NABUJITO* shows the boundaries of the *BENZO*, yet we are able to interact with a VM internal component.

BENZO: LOW-LEVEL GLUE IN PHARO

Contents

3.1	Background	29
3.2	The BENZO Framework	31
3.3	BENZO in Practice	39
3.4	Performance	43
3.5	Extending the Language Runtime	44
3.6	Limitations of Benzo	50
3.7	Conclusion and Summary	54

Introduction

In this chapter we present BENZO a framework developed by Igor Stasenko that connects the low-level VM world with a reflective and dynamic language-side library. Unlike more classical approaches BENZO does not resort to vast set of customized VM primitives or plugins. Instead it relies on a generic primitive to activated native code that is generated at language-side.

BENZO provides a unique experience of being low-level yet using high-level concepts at the same time. This is possible since the framework is implemented at language-side and tightly integrated into the PHARO development environment. In this chapter we present in detail how BENZO interacts with PHARO and what the difficulties are. The key components of BENZO are:

- A generic primitive to activate native code,
- ASMJIT A language-side assembler,
- A language-side library for installing and activating native code.

Based on BENZO we outline 3 unique applications in Section 3.3:

- Foreign Function Interfaces (in more detail in Chapter 4)
- Dynamic Primitives (in more detail in Chapter 5)
- Language-side JIT (in more detail in Chapter 5)

3.1 Background

High-level low-level programming [25] encourages to use high-level languages such as JAVA to build low-level execution infrastructures or to do

system programming. It is successfully used in experimental high-level self-hosted virtual machines (vms) such as JIKES [3]. Frampton et al. present a framework that is biased towards a statically typed high-level language, taking strict security aspects into account. Their approach promotes to address low-level system programming tasks with the tools and abstractions of high-level languages. However, their solution has reduced applicability in a dynamic and reflective context. By reflective, we refer to the combined capabilities to inspect (introspection) and change (intercession) the same execution concepts at runtime [34].

From a reflective point of view it seems natural to dynamically modify the vm at runtime and not just at compile-time. If we are able to modify the vm from language-side we blur the line between these two distinct worlds, becoming indistinguishable to talk about the vm or the language-side. Hence throughout this chapter we use the term language runtime to refer to the running vm combined with the language-side application.

3.1.1 Requirements

Extending the vm is only one particular case of modifying or extending the complete language runtime. Language-side libraries, reflective capabilities, vm extensions or hybrid approaches are other possibilities which we discuss in detail in Section 3.5. All these typical extension mechanisms are not sufficient if we want to modify the vm from language-side, or in our terminology, to reflectively modify the language runtime. Furthermore these mechanisms are based on the fact that there is a clear barrier between language and vm. A solution that crosses these barriers requires the following properties:

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

3.1.2 BENZO a Framework for Reflective High-level Low-level Programming

High-level low-level programming is a powerful technique for system programming without resorting to static low-level environments [25,55] that almost fulfills our requirements. However, in a reflective setup it fails to comply with the first requirement mentioned in the previous paragraph: it does not allow reflective changes at runtime. Our approach for overcoming this limitation consists of BENZO, a lightweight and reflective framework that dynamically generates native code from language-side and allows its execution on the fly. It relies only on a small set of generic vm extensions described in

Section 3.2.1, whereas the vast majority of the framework is implemented as a language-side library.

BENZO originally evolved from the work done by Igor Stasenko on the NATIVEBOOST which we will describe in more detail in the following Chapter ???. Initially BENZO itself was not considered as a separate application but a core of the FFI framework. We extracted the core concepts — dynamic high-level low-level programming — into a separated framework, BENZO.

3.1.3 BENZO Applications

In Section 3.3 we advocate the contribution of BENZO by providing three different incremental examples that heavily use the framework. Unlike typical implementations that would focus on writing them as VM extensions, we implement them completely at language-side using BENZO:

Language-side FFI A complete language-side Foreign Function Interface (FFI) implementation, described in Section 3.3.1 and in more detail in Chapter 4.

Dynamic Primitives A language-side compilation toolchain that replaces system primitives at runtime with customized code, described later in Section 3.3.2 and in more detail in Section 5.1.

Language-side JIT Compiler A JIT compiler that works at language-side and interacts with the VM for code synchronization, described in Section 3.3.3 and in more detail in Section 5.2.

Illustrated by these three distinct examples, the contributions of this chapter are:

1. A *reflective* high-level low-level programming framework that encourages the extension of high-level language runtimes on the fly without the overheads imposed by pure high-level solutions.
2. 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.

3.2 The BENZO Framework

BENZO is implemented in PHARO¹, a SMALLTALK inspired language. PHARO comes with all the reflective capabilities known from SMALLTALK where most language-side components can be altered dynamically. BENZO is implemented at language-side and only requires the help of two simple and

¹<http://pharo.org/>

generic primitives to activate native code and resolve the entry point address position of referenced C functions.

3.2.1 VM Context

PHARO emerged from the Squeak project [28]. The PHARO VM (Cog) implementation [38] also evolved from the original Squeak bytecode interpreter. The current VM uses a moving Garbage Collector (GC) with two generations and uses a JIT that 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 implementation of the Cog VM. Mainly our requirement is being able to generate executable code and activate it at runtime. This is general and essential so it applies to any VM that wants to support dynamic code execution managed at language-side.

Executable Memory. We chose to follow a very lightweight approach to dynamically execute native code at runtime. Since we use PHARO as our host language it is a natural choice to manage the native code at language-side and use as few VM features as possible. Hence we use normal PHARO objects to hold the generated native code.

However, by default the object memory is not executable. This leaves two choices, either mark the whole object memory executable or 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.

The GC of the PHARO VM uses a moving semi-space approach with two generations. Additionally there is a fixed sized executable region used for the JIT as a buffer for runtime generated native code. The JIT space uses its own small garbage collection strategy which is decoupled from the rest of the object memory. This also means that the JIT space does not hold normal PHARO objects but special low-level structures. As mentioned before, the JIT space is limited in size and eventually fills up, causing the JIT to spill older code structures from there.

The JIT-space is built for holding native code objects. However, since the JIT objects are volatile this is not the place to keep long-living language-side objects holding native code. Instead we opt for the completely executable object memory option and store all the executable code in standard PHARO objects. As the VM has a moving GC, it gives us certain restrictions on what kind of native code we can run directly from the language-side. As we will describe in Section 3.2.1, we can access high-level PHARO objects only via an indirection from low-level code.

vm Interaction. The standard way in PHARO to execute low-level code is to use a tag in the method definition. The following example shows the multiplication 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 3.1-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 PHARO code in the method body. Figure 3.1-b describes it. In the floating point multiplication example the fallback code uses a slow conversion method to polymorphically convert other objects to floats and defer the multiplication.

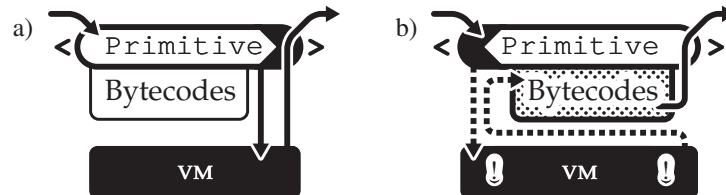


Figure 3.1: Generic primitive methods in PHARO: a) By default a primitive completely bypasses the bytecode, b) A failing primitive executes the bytecode as fallback.

BENZO uses the primitives as a gate to enter the low-level world from the language-side. Our custom primitive then executes the generated native code and returns to language-side. This 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 3.2 shows the structure of a PHARO 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 PHARO 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.

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 PHARO method. The code example `interrupt3` shows a very basic application of our infrastructure.

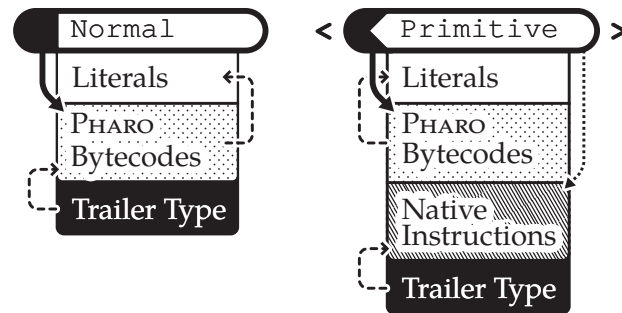


Figure 3.2: A standard PHARO compiled method on the left and a method with appended native instructions generated by BENZO.

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

Code Example 3.1: PHARO method using BENZO for very basic low-level debugging.

The primitive named `primitiveNativeCall` on the first line tries to run the native instructions appended to the compiled method. When there is no native code yet the primitive fails and on return it evaluate the rest of the PHARO code in the method. In Section 3.2.2, through more detailed examples, we describe how BENZO uses PHARO code to generate the native instructions. Figure 3.3 shows the resulting compiled method in full detail.

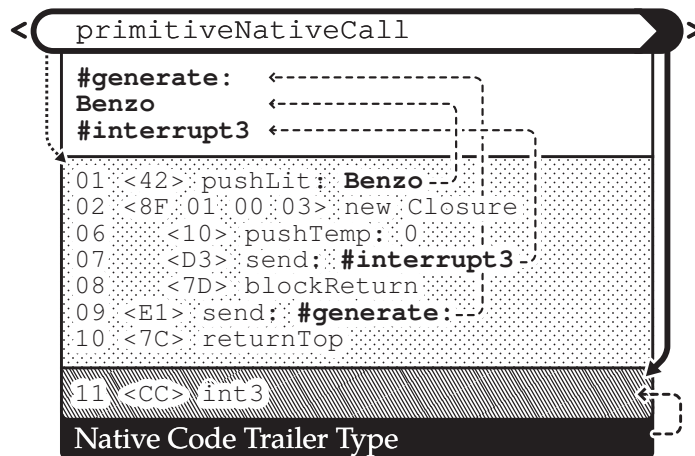


Figure 3.3: Example of PHARO method with native code which calls the low-level debug exception handler `INT3`. The bytecode references objects such as symbols for messages sends indirectly via the literal frame residing at the beginning of the method.

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 do not match, BENZO signals a failure that causes the VM to evaluate the fallback PHARO 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.

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 PHARO objects when they are in the method 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 change 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.

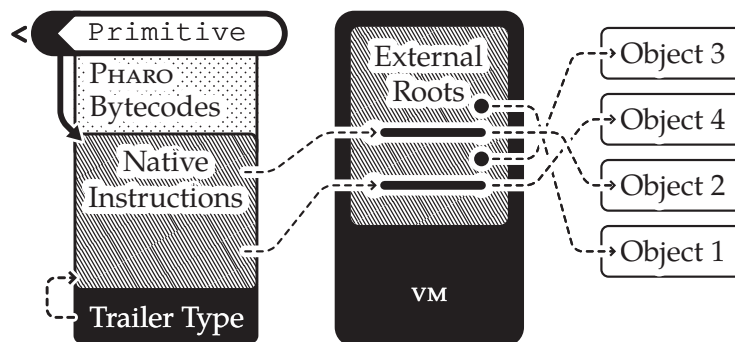


Figure 3.4: Pointers to objects registered as external roots are pinpointed at fixed offset in global VM-level object.

BENZO 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 PHARO objects. Summarizing, for accessing PHARO 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 3.4. BENZO only adds an `Array` to the external root objects which is managed from language-side and administers all references.

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, the managed execution 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. These context switches impose an overhead and can be avoided in the case of calling native code. For this reason 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. Figure 3.5 shows how the native code from a BENZO enabled method

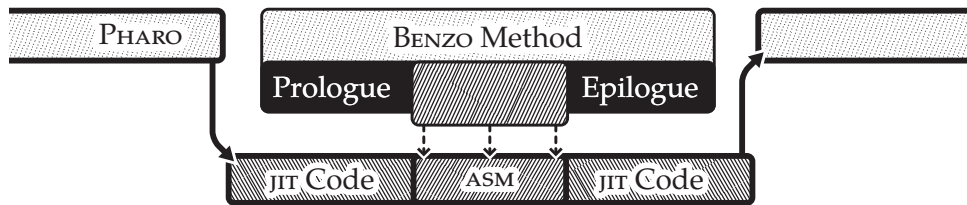


Figure 3.5: BENZO inlining language-side native code into jitted mode.

is inlined into the JIT infrastructure. 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 3.4.

Error Handling. BENZO provides an error handling facility that allows one 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. The following code shows an example application of this behavior.

```
failWithErrorMessage
  <primitive: 'primitiveNativeCall'
  module: 'Benzo' >
  Benzo x86 generate: [ :asm :helper |
    helper failWithMessage: 'Value is not 0' ].
```

Under the hood, BENZO reuses the existing built-in error mechanism of PHARO primitives. However, primitives only allow for error number to be returned which limits the expressiveness of the error messages. To circumvent this limitation BENZO assigns a unique error number for each error message pass to `failWithMessage:.` The mapping between the two error message

representation, its error number and the message string itself, happens at language-side. BENZO simply reuses the existing infrastructure to improve the debugging tasks and promote a better interaction with developers.

3.2.2 BENZO'S Language-Side Implementation

As a key design decision, we determine to keep the interface to the low-level world minimal. The following describes the features of BENZO at the high-level language-side.

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

```
Benzo x86 generate: [ :asm |
    asm mov: 1 asUImm to: asm registers 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. The full features of the high-level environment 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. It is worth noting that all are standard message sends:

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

The vm helper exposes a basic, low-level interface to access objects and its properties. Additional methods cover the access to the external roots described in Section 3.2.1. 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 language-side. On x86 this defaults to EAX. In Section 3.3 we introduce more substantial applications based on BENZO.

Code Activation. So far we only broadly described how BENZO activates the native code. In a nutshell, we generate native code using our own language-side assembler and then we attach the native instructions to compiled methods as shown in figure Figure 3.3. Additionally we mark the method to use

a primitive defined BENZO plugin. The BENZO primitive is responsible for the native code activation which consists of three main steps:

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

The first time a method with BENZO-based native code is activated the linked BENZO primitive will fail and run the normal PHARO code in this method (see Section 3.2.1). This is where the actual native code generation happens. As shown in previous examples, the native code is expressed in standard PHARO code using our language-side assembler. Once the whole code is generated, it is appended to the compiled method body leaving the existing PHARO byte-codes intact. Behind the scenes BENZO adds some more information to the code as the previously mentioned platform marker. After the native code is installed in the compiled method, we still run PHARO code to restart the same BENZO-enabled method again. For clarification we visualize the code activa-

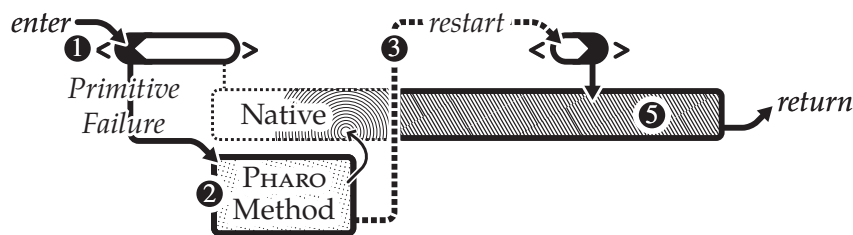


Figure 3.6: Native code activation with BENZO: The first call triggers the code generation written in PHARO. Then the method is restarted and the cached native code executed.

tion process in Figure 3.6 and the following list describes the separate steps in more detail:

Activation: In the first step (cf. ❶) the BENZO primitive is activated. The primitive checks if the compiled method actually contains native code.

Code Generation: On the first activation there is no native code available yet. Hence the primitive will fail and the PHARO body (cf. ❷) of the BENZO-enabled method gets evaluated. This is where we use the BENZO API and write native instructions as shown in previous examples.

Code Installation: After installing the native code in the method trailer, the BENZO-enabled method is reactivated with the original arguments (cf. ❸). For activation BENZO uses PHARO's `#perform:withArguments:` to reflectively restart the method.

Method Reactivation: Again we end up in the BENZO activation primitive (cf. ④). However, this time native code is present and thus the BENZO jumps to native code attached to the compiled method (cf. ⑤) and returns the generated result.

3.3 BENZO in Practice

In this section we present the outline 3 distinct solutions built on top of BENZO: A FFI, dynamic primitives and a JIT (Chapter 4 and Chapter 5 provide more detailed insight). Typically these implementations would require a custom-tailored VM or specialized plugins. However, we show that it is possible to implement them using the generic functionality provided by BENZO.

The chosen applications, starting with the FFI, are increasingly more VM bound. Whereas the typical FFI implementation is based on an separate plugin, dynamic primitives or a JIT require persistent changes in the underlying VM.

3.3.1 NATIVEBOOST: BENZO-based Foreign Function Interface

FFIs enable a programmer to call external functions without the need to implement additional VM extensions. NATIVEBOOST [14] is a production-ready FFI for PHARO, developed on top of BENZO. For a detailed discussion of the implementation of NATIVEBOOST see Chapter 4. An FFI implementation consists of two main parts: calling external functions and converting data between the two environments. Typically a big percentage of these two parts are implemented at VM-level with statically defined bindings. Relying on BENZO's 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.



Figure 3.7: NATIVEBOOST generates native code at language-side with BENZO to perform an FFI callout to an external function.

A very simple example to illustrate the functionality of NATIVEBOOST is to access the current environment variables with the `getenv` C function. `getenv` takes a name as single argument and returns the value of that environment variable as a string:

```

getenv: name
  ^ NativeBoost call: 'String getenv(String name)'

```

In this example `NATIVEBOOST` automatically detects, using reflection, that the argument for the `PHARO` method corresponds to the one of the low-level C function. The most important aspect about this example is that it is written with standard `PHARO` code, a property that extends to almost the complete implementation. `NATIVEBOOST`, additionally to the native code activation, relies on two simple primitives provided by `BENZO` to retrieve addresses of external functions (`dlsym`) and to load external libraries (`dlopen`).

`NATIVEBOOST` generates the glue code to call external functions dynamically at run time. It relies on `BENZO`'s features presented in Section 3.2.2 to generate and activate native code at runtime. This gives `NATIVEBOOST` a significant advantage over static approaches: the generated native code is specific to the callout. For instance in the `getenv` example, the marshalling code for converting from the internal `PHARO` strings to C strings is written a small assembler routine. In this specific context, the assembler code is faster than any language-side code. Yet `NATIVEBOOST` is very flexible since all these conversion routines are defined at language-side. Each language-side library can define its own highly efficient conversion routines for types that are used in `FFI` callouts, which is not directly possible to do with a `VM` extension.

3.3.2 Reflective Primitives

`WATERFALL` is the second application we use to validate `BENZO`. This project has been developed by Guido Chari and allows a programmer to dynamically replace primitives at language-side, essentially reflectively altering the `VM`. This is a step further than previously presented `BENZO`-based `FFI` which allows us to call external functions by generating the callout code at language-side. From an abstract point of view we replace language-side methods with native routines. `NATIVEBOOST` does not directly synthesize new features but only makes external functionality available to the language itself.

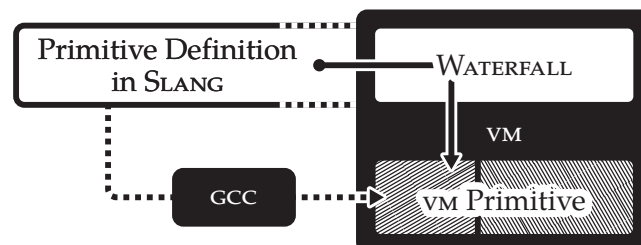


Figure 3.8: `WATERFALL` reuses the definitions of `VM` primitives written in `SLANG` and compiles them dynamically. The same primitive definition is used for generating the static primitive at `VM` compilation time.

As explained previously in Section 3.2.1 BENZO uses PHARO's primitives to activate native code. Since PHARO is an open system we can extend this behavior to existing methods. Instead of simply adding new methods which call native code we present WATERFALL, a solution that modifies existing primitive methods and replaces them with BENZO-based native code. Instead of manually generating the sources for the primitives we reuse existing code. The VM used for PHARO is metacircular, the VM sources are written in the same language, in our case in a simplified subset of PHARO called SLANG. Hence, the complete definition of the VM including the primitives can be made accessible at language-side by loading the VM sources. WATERFALL then takes the primitive definition written in SLANG and compiles it to native-code.

As Figure 3.8 illustrates, WATERFALL extends the lifetime of the metacircular VM definition to the actual language runtime. By default the primitive definitions written in SLANG are only used to generate the VM source in an intermediate step. A C-compiler such as GCC generates the final binary. By doing so the high-level primitive definitions are absorbed by the intermediate compiler infrastructure. The final binary has no reflective capabilities anymore. From within PHARO we can only activate primitives but the abstract definition is no longer accessible. Hence, we can not directly modify primitives directly without the original VM sources loaded.

WATERFALL provides a complete metacircular infrastructure for primitives. We use WATERFALL to modify primitives on the fly. For instance it becomes possible to instrument the crucial `basicNew` primitive, something that is almost impossible to achieve with pure language-side reflection. Since this primitive is used for object creation, each attempt to monitor this primitive is doomed. If the monitoring code itself would create a new object, infinite recursion would be inevitable. In Section 5.1 we explain in more detail the difficulty of such a task along with promising performance evaluations.

3.3.3 NABUJITO JIT Compiler Prototype Outline

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 PHARO methods. Primitives tend to be more low-level, whereas NABUJITO focuses on high-level PHARO code. The PHARO VM (originally COG VM²) already comes with a JIT that translates bytecodes to native instructions. It transforms PHARO methods into slightly optimized native code at runtime. The most complex logic of the JIT infrastructure deals with the dynamic nature of the PHARO environment. Methods

²<http://www.mirandabanda.org/cog/>

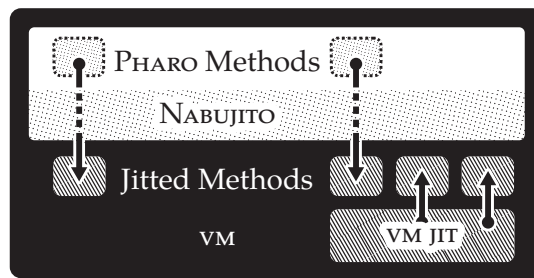


Figure 3.9: NABUJITO compiles standard PHARO methods with the help of BENZO to the same format the VM JIT uses.

and classes can be changed at runtime and that has to be addressed by the JIT infrastructure. This implies that an efficient JIT infrastructure needs substantial access to language-side structures; in our case classes, methods. This information is readily accessible in PHARO through the standard reflective API. However, at VM-level this requires more effort, and thus imposes strong requirements on the design of classes and methods at language-side. The JIT infrastructure is a hybrid between VM logic and language-side reflection.

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 JIT infrastructure. There exists more important stages as an additional register allocation pass to reduce the number of stack operations [37,38]. The existing JIT infrastructure is implemented in SLANG [9, Ch. 5] as the rest of the VM. We believe that a hard-coded static and low-level implementation is not optimal for several reasons:

- Optimizing PHARO code requires strong interactions with the dynamic environment.
- Accessing language-side properties from the VM-side is hard.
- Changing the JIT compiler requires changes at VM-level.
- The JIT reimplements primitives for optimization reasons resulting in code duplication.

Implementing NABUJITO with BENZO. Motivated by the aforementioned implications of a VM-level JIT we conceived NABUJITO a prototype JIT compiler based on BENZO. 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 as a visitor over the existing intermediate bytecode representation. Additionally we reimplemented vital native routines for the JIT which are not directly exported by the VM using BENZO. NABUJITO relies on the following VM-level infrastructure to manage and run native code for any PHARO method:

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

Dynamic Code Generation. For standard methods `NABUJITO` takes the bytecodes and transforms them to native code. It also applies optimizations such as creating low-level branches for `PHARO` level branching operations like `if-True:`. 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`. Section 5.2 gives a more detailed insight of the design and performance of `NABUJITO`.

3.4 Performance

In this section we discuss the general performance characteristics of `BENZO` for the three example applications outlined in the previous section. A more detailed validation is presented later in Section 4.3 (`FFI`), Section 5.1.4 (`WATERFALL`) and Section 5.2.3 respectively.

One-time Code Generation Overhead. `BENZO` allows the generation of specialized and thus efficient native code. In Section 3.2 we explained how `BENZO` causes only a one-time overhead for native code generation. Thereafter it is cached for later activations. The three use case presented in Section 3.3 heavily benefit from this fact. Generating code at language-side poses a significant overhead compared to invoking a precompiled native implementation. However, this is only a one time overhead. For instance the `BENZO`-based `FFI` implementation presented in Section 3.3.1 outperforms a `VM`-level `FFI`-plugin due to a more flexible language-side implementation which generates specialized code for each `FFI` callout. These results are shown in the following Table 3.1.

As an example performance evaluation we present is for `NATIVEBOOST` the `BENZO`-based `FFI`. Compared to a static plugin-based `FFI` implementation `NATIVEBOOST` has only a one-time startup overhead. 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 certain cases the penalty for the language-side code generation of `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. An in-depth evaluation of NATIVEBOOST comparing against other solutions is presented later in Chapter 4. The following table contains a performance comparison of three different FFI implementations for PHARO that represents the typical use case.

	Call Time [ms]	Relative Time
NATIVEBOOST	65.34 ± 0.46	1.00
ALIEN	175.77 ± 0.62	≈ 2.69
C-FFI	148.77 ± 0.42	≈ 2.27

Table 3.1: Different FFI implementations in PHARO evaluating `abs(int)`. ALIEN does marshalling at language-side while FFI does everything in VM plugin written in C.

Table 3.1 measures the accumulative time of 100'000 FFI calls. Included in these numbers is at least one additional PHARO message send to activate the NATIVEBOOST method containing the actual call to the C function. NATIVEBOOST outperforms the existing language-side FFI (ALIEN) and the implementation (C-FFI).

The existing language-side C-FFI has a generic plugin to call C-functions and performs type-conversions at language-side. However, converting PHARO objects from and into low-level data is comparably expensive. In NATIVEBOOST this happens directly in custom generated native code and is thus significantly faster. The plugin-based C-FFI is also slower than NATIVEBOOST since it still has generic conversion function for PHARO objects, albeit written in C and thus faster than in ALIEN. However, NATIVEBOOST custom tailored ASM code is still faster than the hard-coded C counterpart.

This simple FFI evaluation already highlights the core benefit of BENZO to generate very customized native code when needed. Yet we have to emphasize that NATIVEBOOST is based on the BENZO infrastructure whereas the other solutions require both a VM plugin whose sole purpose is to enable the FFI functionality. Furthermore NATIVEBOOST benefits from the JIT interaction described in Section 3.2.1. This optimization is especially an important optimization factor when calling out small helper routines where the context switch from jitted mode is no longer negligible.

3.5 Extending the Language Runtime

In the context of BENZO we see a variety of related work spawning different abstraction levels. On a more abstract scale BENZO allows for a new way of extending the complete language runtime, hence we classify the related work

according to the following categories shown in Figure 3.10: general language-side extensions, extensions using reflection, VM-level extensions, and hybrid approaches.

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 language 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 extensions in this context there exist some known approaches:

Language-side Library based on implementing a new or existing library.

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 3.10.

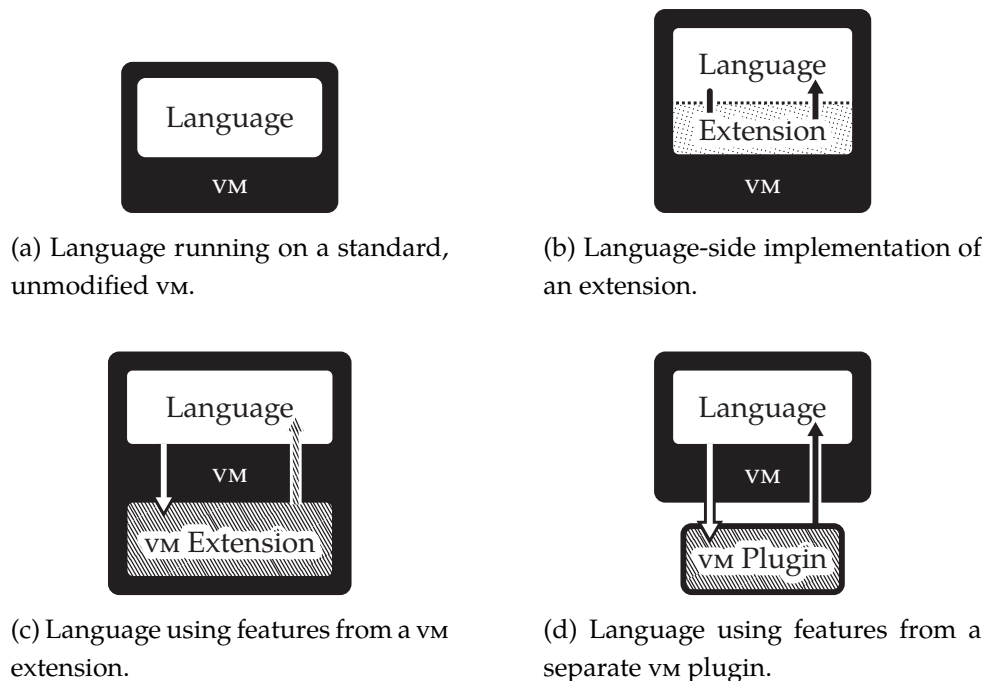


Figure 3.10: Comparison of different extension mechanisms

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

3.5.2 Language-side Reflective Extensions

This is a specialization of the previous approach but in the context of reflective environments. For instance, Meta Object Protocols (MOP) [31] based on reflection [34] 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 PHARO, SMALL-TALK, PYTHON³, Ruby⁴, and others provide reflective capabilities with different depths [4,49]. 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 [35]. Nevertheless, there are exceptions for a subset of reflective behavior which are implemented efficiently using a high-level MOP [53]. 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.

³<http://python.org/>

⁴<http://www.ruby-lang.org/>

3.5.3 VM Extensions / Plugins

Another approach to extend add new features to a programming language is to extend the VM. At this level we differentiate between two extension mechanisms: VM plugins and modified VMs itself. Plugins are direct bindings to external libraries described at VM-side or libraries linked to the VM executable [9, 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. Typically, even very complex VMs such as the SELF VM [47] provide a clean interface to write plugins. Missing high-level concepts make them harder to maintain than language-side code. Depending on the added feature the plugin interface may prove to be too inflexible and the only choice is an modified VM. Such a fork comes at an even higher cost of maintenance. In the following paragraph we will explain the these two main limitations we found when working with plugins: a certain abstraction mismatch compared to host language and the limitations imposed by a clean plugin interface.

Problem: Abstraction Mismatch

Plugins are commonly written in the same language as the VM, at a low abstraction level. The abstraction difference compared to a language-side extensions makes it difficult for standard programmers to modify or write VM plugins. Few exceptions are self-hosted languages [41,48,55]. To support a fluent development process, VMs should come with an infrastructure for building extensions at same abstraction level as the provided language. Instead VMs tend to be rather complex which includes the whole development cycle. For example, only a few VMs have high-level debugging facilities [28,48,55]. It is more common to be stuck in the compilation step waiting for the debugging binary to be ready.

VM Generation Frameworks. The lack of abstraction for VM-level extensions can be addressed by 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 VM⁵ (former JALAPEÑO) [3]. It uses JAVA to metacircularly define a JAVA environment which then generates the final VM. A similar framework is PyPy⁶ [41] 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 [12]

⁵<http://jikesrvm.org/>

⁶<http://pypy.org/>

or PROLOG. However, its main focus lies on an efficient JIT generator mainly for a PYTHON interpreter, and not on a direct, language-side assembler interface. PyPy encourages to use reflection at compile-time which helps to write a maintainable code base.

Problem: Plugin Limitations

Once a programmer is fluent at VM-level a clean plugin interface is a big aid in terms of maintenance. However, certain functionality can not be added by simply creating a separate plugin that encapsulates the new feature. Prominent examples are JIT support, immutability or first-class handles [6]. In all these examples core pieces of the VM have to be modified: the VM has to be forked. From a VM maintenance point of view, forks have to be avoided if possible and should only be used for critical performance issues that can not be properly addressed at language-side or with plugins.

From our experience with PHARO, even promising VM experiments are not maintained for a long time. An example for that is the modified VM supporting back-in-time debugging implemented for an early version of PHARO and SQUEAK [33]. The features would improve debugging without a doubt. However, the VM evolved in the mean time adding new features required by newer versions of PHARO. As a result the back-in-time enabled VM is no longer compatible with PHARO. Presumably it would be easier to port the back-in-time debugger to a new PHARO version if it were implemented purely at language-side.

High-level Low-level Programming. High-level low-level programming [25] 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 [10] 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 language-side application does not allow for reflective extensions of the language runtime. Finally, they do not support the execution nor generation of custom assembly code in the fly.

VM extensions provide good performance at the cost of maintainability. Moreover this approach implies resorting to pure low-level development where tools and abstraction advantages from high-level languages are restricted.

3.5.4 Hybrid Extensions

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 a 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 [24,40].

Other high-level languages such as LUA leverage FFI performance by using a close interaction with the JIT. LUAFFI⁷ for instance is an efficient LUA implementation that inlines FFI calls directly into the JIT compiled code. Similar to BENZO this allows us to minimize the constant overhead by generating custom-made native code. LUAJIT 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 3.5.3.

Kell and Irwin [30] 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.

QUICKTALK [8] follows a similar approach as the dynamic primitives in 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

⁷<https://github.com/jmckaskill/luaffi/>

primitives.

A notable exception to the metacircular vms mentioned earlier is the SELF implementation KLEIN [48]. Unlike typical other metacircular approaches it does not strictly separate compile-time and runtime. The reified vm concepts are available at runtime, which is a result from implementing the typical vm pieces at language-side. Compared to our approach, KLEIN's bridging efforts are much more complete. However, KLEIN is built on a completely new vm infrastructure, whereas BENZO requires only few changes to achieve its functionality.

Hybrid extension are the most promising. They allow for a seamless interaction from high-level language-side to low-level functionality. However, most existing solutions target only specific use cases and can not be reused for other applications.

3.6 Limitations of Benzo

In this section we will point out the current limitations of the BENZO framework. As we outlined in Section 3.4 BENZO is a promising alternative to building vm-plugins. Our prototype use cases, the dynamic primitives and the language-side JIT compiler, even suggest that BENZO can be used as a replacement for certain vm-level modifications. However, throughout the development of these tools we also noticed certain drawbacks and limitations of the current BENZO implementation. We will now discuss the following three main issues in more detail: robustness, debuggability and portability.

3.6.1 Robustness

The first immediately visible flaw of BENZO is that there is currently no support for running the native code in a protected environment. BENZO directly transfer the execution context to the generated native code without protection. Whilst this makes sense from a performance point of view for a stable piece of native code, it is nuisance during development. The most common errors we noticed during development were cause due to stack misalignment and access to invalid memory region. The latter one is also typical for C development, whereas creating a misaligned stack in pure C is not possible.

```
Benzo x86 generate: [ :asm |
    "Push the EAX register on the stack"
    asm push: asm EAX ]
```

Code Example 3.2: Simple BENZO code possibly leading to an unbalanced stack.

```
Benzo x86 generate: [ :asm |
    "Puhs the value of the location EAX points to
    asm push: asm EAX ptr ]
```

Code Example 3.3: BENZO code possibly leading memory access violation.

In PHARO it is possible to willingly corrupt the stack by for instance injecting a wrong sender as illustrated with following the code example.

```
"Store an invalid object in a context variable used
by the JIT and the VM"
thisContext instVarNamed: #closureOrNil put: 1.
```

The context is exposed as first-class object to the language and thus we can reflectively modify it. However, at the same time there it has to fulfill a contract with the JIT which maps it to a native stack frame for performance reasons. In the current COG VM reflective accesses to the current context/stack frame are not checked. Hence we can use it to inject a wrong object, in our case a small integer. This would require additional checks at the JIT to properly read the small integer. However, for performance reasons this is not done, resulting in a wrongly read value and a subsequent crash of the VM.

In BENZO the standard developer will most probably be more familiar with PHARO than with C or even Assembler, which will easily lead to the aforementioned errors. Which means that we have to be prepared for these common errors. Currently the active operating system process will be killed when the misaligned stack eventually lead to an invalid memory access. From a PHARO point of view this is unacceptable behavior. Typically every error is revealed in PHARO by opening a debugger, giving the programmer a chance to figure out the problem at hand. Enabling debugging support for the low-level BENZO code is not that easy as we will show in the following Section 3.6.2.

As a first step we should provide a debug mode for BENZO where the low-level errors do not terminate the main process. The most simple way to achieve this is by forking the whole VM process at the moment the native BENZO code is activated. However, the current implementation of the PHARO VM does not support clean forking, even so close implementations such as SQUEAK supports forking with a specific VM plugin⁸. Implementing a debugging version of the BENZO plugin to activate native code in a forked process would solve this issue, though at the cost of an additional primitive and the use of an old VM.

Addressing faulty stack management requires more effort. One possibility is to use an existing ASM simulation framework and run the code in there and check for unbalanced stack operations. This difficulty leads us the second issue with BENZO.

⁸<http://wiki.squeak.org/squeak/708/>

3.6.2 Low-level Debugging

The second main limitation of BENZO is the lack of a dedicated debugger. PHARO inherits the long standing tradition of SMALLTALK to provide an excellent debug interaction for programmers.

The PHARO debugger shows the stack, the current receiver along with the temporary variables and argument. However when working with BENZO this important tool is no longer available. Currently the only way to debug BENZO code is to launch the VM upfront in a C debugger such as GDB or LLDB. However, the debug interaction is rather limited compared to PHARO. Not only does the programmer have to resort to an external tool, but there are only a handful mostly proprietary standalone debuggers available.

For a better low-level debugging experience we have to rely on a complete IDE such as ECLIPSE or XCODE, a rather cumbersome overhead to standard PHARO programming.

Besides using these external tools there is no simple alternative. The only way to provide a seamless debugger is to either build on the fork solution presented in the previous Section 3.6.1 or a separate simulator. System libraries such as PTRACE enable debugging for an external process. It would be possible to write bindings to this library with an FFI from within PHARO and debug a BENZO-enabled method this way.

Even so this would be a great step forward compared to the current infrastructure it implies that the programmer anticipates debugging. Something that is very uncommon in PHARO as the debugger is tightly integrated into the standard development environment. A potential solution to this problem would be to install low-level signal handlers which try to backtrack from signals triggered by memory access violations such as SIGBUS and SIGSEGV under Linux.

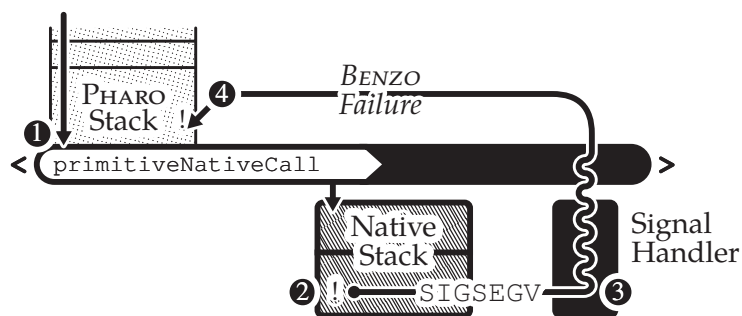


Figure 3.11: Outline of a low-level debugging mechanism to transform low-level errors into a high-level exception using a designated signal handler which walks the stack until it finds the `primitiveNativeCall` activation.

The following list explains the detailed steps of Figure 3.11.

1. Standard PHARO method activating a BENZO-enabled method through the `primitiveNativeCall` primitive.
2. Native code causing a memory access violation (for example SIGSEV) which can not be handled by PHARO directly.
3. Low-level signal handler is activated by the operating system and tries to walk back the native stack up to the `primitiveNativeCall` activation.
4. After successfully finding the `primitiveNativeCall` the signal handler sends a BENZO failure back to PHARO.

We assume that with the outlined mechanism an important fraction of the occurring memory access violations could be handled adequately in PHARO. However, since there is no real protection in the native code, there is no guarantee that the main PHARO VM can continue working. Faulty native code might have corrupted the main PHARO stack or heap beyond repair. Nevertheless, this scheme could bring approximate a high-level user experience for native BENZO code.

3.6.3 Platform Independence

The previous two issues discussed, robustness and debuggability, both target the integration into the existing PHARO development environment. This third issue on the other hand is related to the system integration: platform independence.

The INTEL x86 instruction set is widely distributed and supported by a variety of operating systems and thus the primary choice as native backend for BENZO. Since the generated code is independent of C functions it works on all platforms with x86 support. However, other architectures such as ARM start to be more widespread and inevitable BENZO has to be ported on other platforms. In contrast to PHARO itself, BENZO does not work on ARM platform since all the low-level code is written in terms of x86 instructions. While there is no way around a ARM version of the underlying assembler, we think that porting all existing BENZO routines is too costly. Instead we suggest to use an intermediate format that is platform independent more high-level than direct assembler instructions. We call this intermediate format VIRTUALCPU and it has the following properties:

- High-level three-address-code (TAC) instructions
- Automatic stack-management

Along with the ongoing work for this thesis we already started implementing VIRTUALCPU as a fork of the original compiler infrastructure of the PINOCCHIO VM.

VIRTUALCPU. In this following paragraph we are going to highlight the benefits and usage of this intermediate format. `VIRTUALCPU` is based on a TAC to simplify the adoption of optimizations such as SSA. These TAC instructions take the following form:

```
result := argument1 OP argument2
```

There are three operands involved, `result`, `argument1` and `argument2`, from which the name of this instruction format originates. Based on this assumption, each standard `VIRTUALCPU` instruction returns a temporary variable which can be used for further operations. The following code example outlines the basic usage of `VIRTUALCPU`:

```
Benzo vcpu x86 generate: [ :asm | | temp1 temp2 |
    temp1 := asm memoryAt: 16r12345.
    temp2 := asm uint: 2.
    asm return: temp1 + temp1 ]
```

Code Example 3.4: Basic `VIRTUALCPU` Example

Which corresponds to the same functionality expressed in the following x86 instructions:

```
Benzo x86 generate: [ :asm |
    asm mov: 16r12345 ptr to: asm EAX.
    asm add: asm EAX with: 2.
    asm return ]
```

To get to the final native instructions the `VIRTUALCPU` infrastructure compiles the high-level instructions to the specific backend. The current compiler is divided into the following passes:

- Platform Specific Transformation
- Register Allocation
- Superfluous Assignment Remover
- Platform Specific Assembler

Applying these compiler passes to the example in Listing 3.4 yields the following native instructions with explicit register assignments:

```
mov    6 -> EDX
mov    2 -> ECX
add EDX,    ECX
mov ECX -> EAX
return
```

With a properly parametrized register allocator and a separate constant folding pass the result could be greatly improved.

3.7 Conclusion and Summary

In this chapter we presented **BENZO**, an integral approach for reflective high-level low-level programming. **BENZO** consists of three core parts: **ASMJIT** a language-side assembler, a set of primitives to activate native code and language-side library to handle dynamic code installation and activation.

BENZO promotes a smooth and powerful interaction with the low-level world by dynamically generating native code from language-side. This enables to exploit the underlying platform capabilities when strongly needed without leaving the host development platform. Most of the **BENZO** infrastructure is implemented at language-side and thus susceptible to modification. As a result, **BENZO** advocates the use of development tools and abstraction level of the high-level language for as much as possible or desired.

BENZO Applications. Based on **BENZO** we outlined in this chapter three example applications: an **FFI** in Section 3.3.1, dynamic primitives in Section 3.3.2 and a language-side **JIT** compiler in Section 3.3.3. Typically all these applications require either a modified **VM** or a dedicated plugin while our implementations are based on a central framework for low-level interaction. In this chapter we only outlined these application to stress the fact that they use **BENZO** and only the following chapters will shed light on the implementation details of these three applications.

BENZO Performance. Using the three **BENZO**-based applications we evaluate the performance of **BENZO** compare to a typical **VM**-level implementation. In summary we note that **BENZO**'s code generation at language-side is slow compared to a single invocation the final native code. However, this is only a one-time overhead since **BENZO** caches native code transparently at language-side. Additionally we generate specific native code for each different application we easily outperform a static solution. This becomes evident with the **FFI** implementation that is based on **BENZO**. Our mature **FFI** implementation outperforms an existing **C-FFI** implementation by a factor of 1.5 even though we control every aspect from language-side.

By combining high-level reflection capabilities 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 our **JIT** compiler poses only a one-time overhead when generating native code.

BENZO's Limitations. Even though **BENZO** allows us to implement the three example applications at language-side, the complete development interaction requires improvement. Currently there is not protection against faulty assembler code nor support for a low-level debugger. To clearly support the

theory of boundary-less low-level interaction a basic debugging infrastructure is required as outlined in Section 3.6.

BENZO Outlook. BENZO shows that promoting clear interfaces for controlling low-level code completely from language-side produces efficient solutions for system programming without resorting to pure low-level solutions. Our set of BENZO-based applications shows that our approach is feasible and efficient. At the first sight BENZO is a simple application to invoke native code but we think that it opens doors for a new kind of language runtime. In this envisioned system there is no longer a clear barrier between VM and language-side. This might seem far fetched but becomes more apparent when having a look at the JIT of PHARO which reimplements a performance critical set of primitives in its own native code. Essentially this is code duplication since the primitives already exist as normal C code in the VM sources. With BENZO and the described dynamic primitives we should reuse the same code base for creating the JIT representation of the primitive.

After presenting the basis of our high-level low-level programming framework in PHARO we will focus on its application in the following chapters.

VALIDATION: FFI

Chapter

4

Contents

4.1	Background	58
4.2	NATIVEBOOST-FFI: an Introduction	60
4.3	Performance Evaluation	68
4.4	Implementation Details	72
4.5	NATIVEBOOST Limitations	74
4.6	Conclusion	79

Introduction

In the previous Chapter 3 we presented `BENZO`, a framework that spans over several abstraction layers to enable low-level programming. `BENZO` is the generic backend for a variety of applications we outlined in the previous chapter: a Foreign-Function-Interface `FFI`, a prototype for dynamic primitives and a prototype for a language-side `JIT` compiler. In this chapter we present the most mature `BENZO` application, the `FFI` which is successfully used in production in `PHARO 2.0` and newer.

`FFIs` are a prerequisite for close system integration of a high-level language. With `FFIs` the high-level environment interacts with low-level functions allowing for a unique combination of features. This need to interconnect high-level (Objects) and low-level (C functions) has a strong impact on the implementation of a `FFI`: it has to be flexible and fast at the same time.

`NATIVEBOOST` is a language-side approach to `FFIs` developed by Igor Stasenko that only requires minimal changes to the `VM`. `NATIVEBOOST` directly creates specific native code at language-side and thus combines the flexibility of a language-side library with the performance of a native plugin.

4.1 Background

Currently, more and more code is produced and available through reusable libraries such as OpenGL¹ or Cairo². While working on your own projects using dynamic languages, it is crucial to be able to use such existing libraries with little effort. Multiple solutions exist to achieve access to an external library from dynamic languages that are executed on the top of a virtual machine (VM) such as PHARO³, LUA⁴ or PYTHON⁵. Figure 3.10 depicts four possibilities of dealing with new or external libraries in a high-level language.

Language-side Library. One solution is to re-implement a library completely at language-side (cf. Figure 3.10.a). Even though this is the most flexible solution, this is often not an option, neither from the technical point of view (performance penalty), nor from the economic point of view (development time and costs).

VM Extension. The second one (Figure 3.10.b) is to do a *VM extension* providing new primitives that the high-level language uses to access the native external library. This solution is generally efficient since the external library may be statically compiled within the VM. However a tight integration into the VM also means more dependencies and a different development environment than the final product at language-side.

VM Plugin. The third solution (Figure 3.10.c) is similar to the previous one but the extension is factored out of the VM as a *plugin*. This solution implies again a lot of low-level development at VM-level that must be done for each external library we want to use. Additionally we have to adapt the plugin for all platforms on which the VM is supposed to run on.

FFI. A higher-level solution is to define *Foreign Function Interfaces* (FFIs) (cf. Figure 3.10.d). The main advantage of this approach is that once a VM is FFI-enabled, only a language extension (no VM-level code) is needed to provide access to new native libraries. From the portability point of view, only the generic FFI VM-plugin has to be implemented on all platforms.

Implementing an FFI library is a challenging task because of its antagonist goals:

¹<http://www.opengl.org/>

²<http://cairographics.org/>

³<http://pharo.org/>

⁴<http://lua.org/>

⁵<http://python.org/>

- it must be flexible enough to easily bind to external libraries and also express complex foreign calls regarding the memory management or the type conversions (marshalling);
- it must be well integrated with the language (objects, reflection, garbage collector);
- it must be efficient.

Existing FFI libraries of dynamic languages all have different designs and implementations because of the trade-offs they made regarding these goals and challenges. Typical choices are resorting purely to the VM-level and thus sacrificing flexibility. The inverse of this approach exists as well: FFIs can be implemented almost completely at language-side but at a significant performance loss. Both these pitfalls are presented in more detail in Section 4.3.

This chapter presents `NATIVEBOOST-FFI`⁶ an FFI library at language-side for PHARO that supports callouts and callbacks, which we present in Section 4.2. There are at least two other existing FFI libraries in PHARO worth mentioning: `C-FFI` and `ALIEN`. Nevertheless, they both present shortcomings. `C-FFI` is fast because it is mostly implemented at VM-level, however it is limited when it comes to do complex calls that involve non-primitive types or when we want to define new data types. On the opposite, `ALIEN` FFI is flexible enough to define any kind of data conversion or new types directly at language-side but it is slower than `C-FFI` because it is mostly implemented at language-side. In essence, `NATIVEBOOST-FFI` combines the flexibility and extensibility of `ALIEN` that uses language-side definition for marshalling and the speed of `C-FFI` which is implemented at VM-level. The main contributions of `NATIVEBOOST-FFI` are:

Extensibility: `NATIVEBOOST-FFI` relies on as few VM primitives as possible (5 primitives), essentially to call native code. Therefore, most of the implementation resides at language-side, even low-level mechanisms. That makes `NATIVEBOOST-FFI` easily extensible because its implementation can be changed at any time, without needing to update the runtime (VM). It also presents a noticeable philosophical shift, how we want to extend our language in future. A traditional approach is to implement most low-level features at VM-side and provide interfaces to the language-side. But that comes at cost of less flexibility and longer development and release cycles. On the opposite, we argue that extending language features, even low-level ones, should be done at language-side instead. This results in higher flexibility and without incurring high runtime costs which usually happen when using high-level languages such as PHARO.

⁶<http://code.google.com/p/nativeboost/>

Language-side extension: Accessing a new external library using `NATIVEBOOST-FFI` involves a reduced amount of work since it is only a matter of writing a language-side extension.

Performance: Despite the fact it is implemented mostly at language-side, `NATIVEBOOST-FFI` achieves superior performance compared to other FFI implementations running `PHARO`. This is essentially because it uses automatic and transparent native code generation at language-side for marshalling.

4.2 NATIVEBOOST-FFI: an Introduction

This section gives an overview of the code that should be written at language-side to enable interactions with external libraries.

4.2.1 Simple Callout

Listing 4.1 shows the code of a regular `PHARO` method named `ticksSinceStart` that defines a callout to the `clock` function of the `libc`. `NATIVEBOOST` imposes no constraint on the class in which such a binding should be defined. However, this method must be annotated with a specific pragma (such as `<primitive:module:>`) which specifies that a native call should be performed using the `NATIVEBOOST` plugin.

```
ticksSinceStart
  <primitive: #primitiveNativeCall
    module: #NativeBoostPlugin>
  ^ self
    nbCall: #(uint clock ())
    module: NativeBoost CLibrary
```

Code Example 4.1: `NATIVEBOOST-FFI` example of callout declaration to the `clock` function of the `libc`

The external function call is then described using the `nbCall:module:` message. The first parameter (`#nbCall:`) is an array that describes the signature of C function to callout. Basically, this array contains the description of a C function prototype, which is very close to normal C syntax. The return type is first described (`uint` in this example⁷), then the name of the function (`clock`) and finally the list of parameters (an empty array in this example since `clock` does not have any). The second argument, `#module:` is the module name, its full path or its handle if already loaded, where to look up

⁷The return type of the `clock` function is `clock_t`, but we deliberately used `uint` in this first example for the sake of simplicity even if it is possible to define a constant type in `NATIVEBOOST`.

the given function. This example uses a convenience method of `NATIVEBOOST` named `CLibrary` to obtain a handle to the standard C library.

4.2.2 Callout with Parameters

```

abs: anInteger ❶
  <primitive: #primitiveNativeCall ❷
    module: #NativeBoostPlugin
    error: errorCode> ❸
  ^ self
    nbCall: #(uint abs(int anInteger)) ❹
    module: NativeBoost CLibrary ❺

```

Figure 4.1: Example of the general `NATIVEBOOST-FFI` callout syntax

Figure 4.1 presents the general syntax of `NATIVEBOOST-FFI` through an example of a callout to the `abs` function of the `libc`. The `abs` : method has one argument named `anInteger` (cf. ❶). This method uses the pragma `<primitive:module:error:>` which indicates that the `#primitiveNativeCall` of the `#NativeBoostPlugin` should be called when this method is executed (cf. ❷). An `errorCode` is returned by this primitive if it fails and the regular `PHARO` code below is executed (cf. ❸). The main difference with the previous example is that the `abs` function takes one integer parameter. In this example, the array `#(uint abs(int anInteger))` passed as argument to `#nbCall`: contains two important information (cf. ❹). First, the types annotations such as the return type (`uint` in both examples) and arguments type (`int` in this example). These types annotations are then used by `NATIVEBOOST-FFI` to automatically do the marshalling between C and `PHARO` values as illustrated by the next example. Second, the values to be passed when calling out. In this example, `anInteger` refers to the argument of the `abs` method, meaning that the value of this variable should be passed to the `abs` C function. Finally, this `abs` function is looked up in the `libc` whose an handle is passed in the `module`: parameter (cf. ❺).

4.2.3 Automatic Marshalling of Known Types

Listing 4.2 shows a callout declaration to the `getenv` function that takes one parameter.

```

getenv: name
  <primitive: #primitiveNativeCall
    module: #NativeBoostPlugin>

  ^ self
    nbCall: #(String getenv(String name)

```

```
module: NativeBoost CLibrary
```

Code Example 4.2: Example of callout to `getenv`

In this example, the `NATIVEBOOST` type specified for the parameter is `String` instead of `char*` as specified by the standard `libc` documentation. This is on purpose because strings in C are sequences of characters (`char*`) but they must be terminated with the special character: `\0`. Specifying `String` in the `#nbCall: array` will make `NATIVEBOOST` to automatically do the arguments conversion from `PHARO` strings to C strings (`\0` terminated `char*`). It means that the string passed will be put in an external C `char` array and a `\0` character will be added to it at the end. This array will be automatically released after the call returned. This is an example of automatic memory management of `NATIVEBOOST` that can also be controlled if needed. Obviously, the opposite conversion happens for the returned value and the method returns a `PHARO` string. This example shows that `NATIVEBOOST-FFI` accepts literals, local and instance variable names in callout declarations and it uses their type annotation to achieve the appropriate data conversion. Table 4.1 shows the default and automatic data conversions achieved by `NATIVEBOOST-FFI`.

Primitive Type	PHARO Type
<code>uint</code>	Integer
<code>int</code>	Integer
<code>String</code>	ByteString
<code>bool</code>	Boolean
<code>float</code>	Float
<code>char</code>	Character
<code>oop</code>	Object

Table 4.1: Default `NATIVEBOOST-FFI` mappings between C/primitive types and high-level types. Note that `oop` is not a real primitive type as no marshalling is applied and the raw pointer is directly exposed to `PHARO`.

Listing 4.3 shows another example to callout the `setenv` function. The return value will be converted to a `PHARO Boolean`. The two first parameters are specified as `String` and will be automatically transformed in `char*` with an ending `\0` character. The last parameter is `1`, a `PHARO` literal value without any type specification and `NATIVEBOOST` translates it as an `int` by default.

```
setenv: name value: value
  <primitive: #primitiveNativeCall
  module: #NativeBoostPlugin>

^ self
  nbCall: #(Boolean setenv(String name,
```

```

                                String value,
                                1)
module: NativeBoost CLibrary

```

Code Example 4.3: Example of callout to `setenv`

Another interesting example of automatic marshalling is to define the `abs` method (cf. Figure 4.1) in the `SmallInteger` class and passing `self` as argument in the callout. In such case, `NATIVEBOOST` automatically converts `self` (which is a `SmallInteger`) into an `int`. This list of mapping is not exhaustive and `NATIVEBOOST` also supports the definition of new data types and new conversions into more complex C types such as structures (cf. Section 4.4).

4.2.4 Supporting new types

The strength of language-side `FFIs` appears when it comes to do callouts with new data types involved. `NATIVEBOOST-FFI` supports different possibilities to interact with new types.

Declaring structures. For example, the Cairo library⁸ provides complex structures such as `cairo_surface_t` and functions to manipulate this data type. Listing 4.4 shows how to write a regular `PHARO` class to wrap a C structure. `NATIVEBOOST` only requires a class-side method named `asNBExternalType:` that describes how to marshal this type back and forth from native code. In this example, we use existing marshalling mechanism defined in `NBExternalObjectType` that just copies the structure's pointer and stores it in an instance variable named `handle`.

```

AthensSurface subclass: #AthensCairoSurface
  instanceVariableNames: 'handle'.

AthensCairoSurface class>>asNBExternalType: gen
  "handle iv holds my address (cairo_surface_t)"
  ^ NBExternalObjectType objectClass: self

```

Code Example 4.4: Example of C structure wrapping in `NATIVEBOOST`

Callout with structures. Listing 4.5 shows a callout definition to the `cairo_image_surface_create` function that returns a `cairo_surface_t*` data type. In this code example, the return type is `AthensCairoSurface` directly (not a pointer). When returning from this callout, `NATIVEBOOST` creates an instance of `AthensCairoSurface` and the marshalling mechanism stores the returned address in the `handle` instance variable of this object.

⁸<http://cairographics.org/>

```

primImage: aFormat width: aWidth height: aHeight
  <primitive: #primitiveNativeCall
    module: #Benzo
    error: errorCode>

  ^self nbCall: #(AthensCairoSurface
    cairo_image_surface_create (int aFormat,
                               int aWidth,
                               int aHeight) )

```

Code Example 4.5: Example of returning a structure by reference

Conversely, passing an `AthensCairoSurface` object as a parameter in a callout makes its pointer stored in its `handle iv` (cf. Listing 4.6) to be passed. Since the parameter type is `AthensCairoSurface` in the callout definition, `NATIVEBOOST` also ensures that the passed object is really an instance of this class. If it is not, the callout fails before executing the external function because passing it an address on a non-expected data could lead to unpredicted behavior.

```

primCreate: cairoSurface
  <primitive: #primitiveNativeCall module: #Benzo >

  ^self nbCall: #(
    AthensCairoCanvas cairo_create (
      AthensCairoSurface cairoSurface))

```

Code Example 4.6: Example of passing a structure by reference

Accessing structure fields. In `NATIVEBOOST`, one can directly access the fields of a structure if needed, even if it is not a good practice from the data encapsulation point of view. Nevertheless, it may be mandatory to interact with some native libraries that do not provide all the necessary functions to manipulate the structure. Listing 4.7 shows an example of a C struct type definition for `cairo_matrix_t`.

```

1 typedef struct {
2     double xx; double yx;
3     double xy; double yy;
4     double x0; double y0;
5 } cairo_matrix_t;

```

Code Example 4.7: Example external type to convert back and forth with the Cairo library

Listing 4.8 shows that the `NBExternalStructure` of `NATIVEBOOST-FFI` can be subclassed to define new types such as `AthensCairoMatrix`. The description of the fields (types and names) of this structure is provided by the `fieldsDesc` method on the class side. Given this description, `NATIVEBOOST` lazily generates field accessors on the instance side using the field names.

```

1 NBExternalStructure
  variableByteSubclass: #AthensCairoMatrix.
3
AthensCairoMatrix class>>fieldsDesc
5   ^ #(  double sx; double shx;
        double shy; double sy;
7         double x; double y; )

```

Code Example 4.8: Example of NATIVEBOOST-FFI definition of an External-Structure

Listing 4.9 shows a callout definition to the `cairo_matrix_multiply` function passing `self` as argument with the type `AthensCairoMatrix*`. NATIVEBOOST handles the marshalling of this object to a struct as defined in the `fieldsDesc`.

```

1 AthensCairoMatrix>>primMultiplyBy: m
  <primitive: #primitiveNativeCall
3   module: #Benzo
  error: errorCode>
5
  "C signature"
7 "void cairo_matrix_multiply (
        cairo_matrix_t *result,
9         const cairo_matrix_t *a,
        const cairo_matrix_t *b );"
11 ^self nbCall: #(void  cairo_matrix_multiply
        (AthensCairoMatrix * self,
13        AthensCairoMatrix * m ,
        AthensCairoMatrix * self ) )

```

Code Example 4.9: Example of callouts using `cairo_matrix_t`

Memory management of structures. Table 4.2 shows a comparison between C-managed and PHARO-managed structures. The first ones are allocated in the C heap. Their addresses are fixed and they are passed by reference during a callout. But the programmer must free them by hand when they are not needed. The second ones are allocated in the PHARO object-memory. Their addresses are variable since their enclosing object may be moved by the garbage collector. They can either passed by copy which is costly or by reference. Passing a reference may lead to problems is the C function stores the address and try to access it later on since the address may changed.

4.2.5 Callbacks

NATIVEBOOST supports callbacks from native code. This means it is possible for a C-function to call back into the PHARO runtime and activate code. We will use the simple `qsort` C-function to illustrate this use-case. `qsort` sorts

	Memory	Address	Marshalling	Constraint
C-managed struct	C heap	fixed	passed by reference	must be freed
PHARO-managed struct	Object memory	variable	passed by reference passed by copy	may move costly

Table 4.2: Choice of Wrapping Structures in NATIVEBOOST

a given array according to the results of a compare function. Instead of using a C-function to compare the elements we will use a callback to invoke a PHARO block which will compare the two arguments.

```
bytes := #[ 120 12 1 15 ].
callback := QSortCallback on: [ :a :b |
    (a byteAt: 0) - (b byteAt: 0) ].
```

```
self ffiQSort: bytes
    length: bytes size
    compareWith: callback
```

Code Example 4.10: Example of callout passing a callback for qsort

Code Listing 4.10 shows the primary PHARO method for invoking qsort with a QSortCallback instance for the compare function. In this example qsort will invoke run the PHARO code inside the callback block to compare the elements in the bytes array.

To define a callback in NATIVEBOOST we have to create a specific subclasses for each callback with different argument types.

```
NBFFICallback
    subclass: #QSortCallback.

NBFFICallback class>>signature
    ^#(int (NBExternalAddress a, NBExternalAddress b))
```

Code Example 4.11: Example of callback definition

Code Listing 4.11 shows QSortCallback which takes two generic external addresses as arguments. These are the argument types that are being passed to the sort block in Example Listing 4.10.

```
ffiQSort: base len: size compare: qsortCallback
<primitive: #primitiveNativeCall module: #Benzo>

"C qsort signature"
"void qsort(
    void *base,
    size_t nel,
    size_t width,
    int (*compar)(const void *, const void *));"
```

```

^ self
  options: #( optMayGC )
  nbCall: #(void qsort (
            NBExternalAddress array,
            ulong size,
            1, "sizeof an element"
            QSortCallback qsortCallback))
  module: NativeBoost CLibrary

```

Code Example 4.12: Example of callout passing a callback

The last missing piece in this example is the callout definition shown in Code Listing 4.12. The `NATIVEBOOST` callout specifies the callback arguments by using `QSortCallback`.

Callback lifetime. Each time a new callback is instantiated it reserves a small amount of external memory which is freed once the callback is no longer used. This is done automatically using object finalization hooks..

4.2.6 Overview of NATIVEBOOST-FFI Internals

This section provides an overview of the internal machinery of `NATIVEBOOST-FFI` though it is not mandatory to know it in order to use it as demonstrated by previous examples.

General Architecture. Figure 4.2 describes the general architecture of `NATIVEBOOST`. Most code resides at language-side, nevertheless some generic extensions (primitives) to the VM are necessary to activate native code. At language-side, callouts are declared with `NATIVEBOOST-FFI` which processes them and dynamically generates x86 native code using the `AsmJit` library. This native code is responsible of the marshalling and calling the external function. `NATIVEBOOST` then uses a primitive to activate this native code.

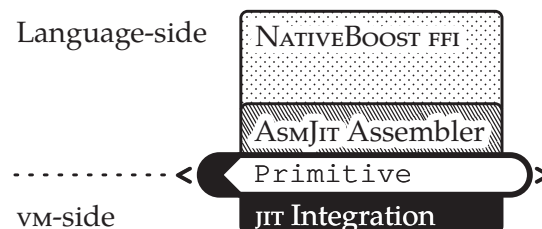


Figure 4.2: `NATIVEBOOST` is implemented in `PHARO` using the language-side assembler and then the `BENZO` primitive to activate native code.

Callout propagation. Figure 4.3 shows a comparison of the resolution of a FFI call both in `NATIVEBOOST-FFI` and a plugin-based FFI. At step 1, a FFI call is emitted. The `NATIVEBOOST-FFI` call is mostly processed at language-side and it is only during step 4 that a primitive is called and the VM effectively does the external call by executing the native code. On the opposite, a plugin-based FFI call already crossed the low-level frontier in step 2 resulting that part of the type conversion process (marshalling) is already done in the VM code. In `NATIVEBOOST-FFI`, doing most of the FFI call processing at language-side makes easier to keep control, redefine or adapt it if needed.

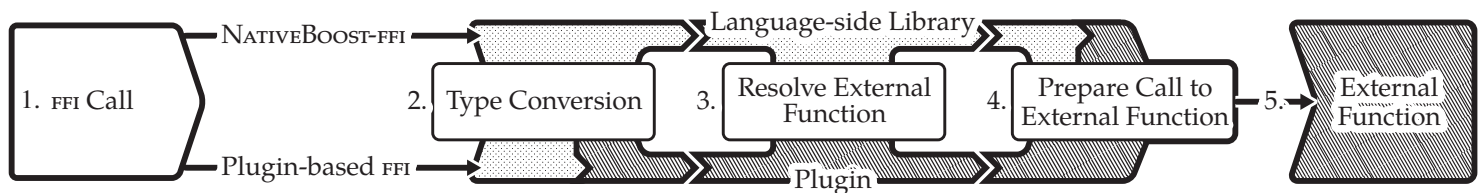


Figure 4.3: Comparison of FFI calls propagation in `NATIVEBOOST-FFI` and a typical VM plugin-based implementation. `NATIVEBOOST` resorts to VM-level only for the native-code activation, whereas typical implementations cross this barrier much earlier.

4.3 Performance Evaluation

In this section we compare `NATIVEBOOST` with other FFI implementations.

ALIEN FFI: An FFI implementation for `SQUEAK/PHARO` that focuses on the language-side. All marshalling happens transparently at language-side.

C-FFI: A C based FFI implementation for `SQUEAK/PHARO` that performs all marshalling operations at VM-side.

LUAJIT: A fast `LUA` implementation that has a close FFI integration with JIT interaction.

Choice of FFI Implementations. To evaluate `NATIVEBOOST` we explicitly target FFI implementations running on the same platform, hence we can rule out additional performance differences. `ALIEN` and `C-FFI` run in the same `PHARO` image as `NATIVEBOOST` allowing a much closer comparison. `ALIEN FFI` is implemented almost completely at language-side, much like `NATIVEBOOST`. However, as the following benchmarks will stress, it also suffers from performance loss. On the other end there is `C-FFI` which is faster than `ALIEN` but by far not as flexible. For instance only primitive types are handled directly. As the third implementation we chose `LUA` which is widely used as scripting language in game development. Hence much care has been taken to closely integrate `LUA`

into C and C++ environments. LUAJIT integrates an FFI library that generates the native code for marshalling and directly inlines C functions callout in the JIT-compiled code.

Evaluation Procedure. To compare the different FFI approaches we measure 100 times the accumulative time spent to perform 1'000'000 callouts of the given function. From the 100 probes we show the average and the standard deviation for a 95% confidence interval in a gaussian distribution. To exclude the calling and loop overhead we subtract from each evaluation the time spent in the same setup, but without the FFI call. The final deviation displayed is the arithmetic average of the measured deviation of the base and the callout measurement.

The three FFI solutions that work in PHARO (NATIVEBOOST, ALIEN, C-FFI) are evaluated on the very same PHARO 1.4 (version 14458) image on a PHARO VM (version of May 5. 2013). For the LUA benchmarks we use LUAJIT 2.0.1. The benchmarks are performed under the constant conditions on a MacBook Pro. Even though a standalone machine could improve the performance we are only interested in the relative performance of each implementation.

Choice of Callouts. We chose a set of representative C functions to stress different aspects of an FFI implementation. We start with simple functions that require little marshalling efforts and thus mainly focus on the activation performance and callout overhead. Later we measure more complex C functions that return complex types and thus stress the marshalling infrastructure.

4.3.1 Callout Overhead

The first set of FFI callouts show mainly the overhead of the FFI infrastructure to perform the callout.

For the first FFI evaluation we measure the execution time for a `clock()` callout. The C function takes no argument and returns an integer thus guaranteeing a minimal overhead for marshalling and performing the callout.

	Call Time [ms]	Relative Time
NATIVEBOOST	492.1 ± 1.4	1.0
ALIEN	606.6 ± 4.8	≈ 1.2
C-FFI	541.7 ± 1.8	≈ 1.1
LUAJIT	343.0 ± 2.4	≈ 0.7

Table 4.3: Speed comparison of an `uint clock(void)` FFI call (see Code Listing 4.1).

`abs` is about the same complexity as the `clock` function, however accepting a single integer as argument.

	Call Time [ms]	Relative Time
NATIVEBOOST	65.34 ± 0.46	1.00
ALIEN	175.77 ± 0.62	≈ 2.69
C-FFI	148.77 ± 0.42	≈ 2.27
LUAJIT ⁹	2.04 ± 0.03	≈ 0.03

Table 4.4: Speed comparison of an `int abs(int i)` FFI call (see Figure 4.1).

Evaluation. For measuring the calling overhead we chose the `abs` FFI callout. This C function is completed in a couple of instructions which in comparison to the conversion and activation effort of the FFI callout is negligible. In Table 4.4 we see that NATIVEBOOST is at least a factor two faster than the other PHARO implementation. Yet LUAJIT outperform NATIVEBOOST by an impressive factor 30. LUAJIT has a really close integration with the JIT and this is what makes the impressive FFI callout results possible.

4.3.2 Marshalling Overhead for Primitive Types

The third example calls `getenv('PWD')` expecting a string as result: the path of the current working directory. Both argument and result have to be converted from high-level strings to C-level zero-terminated strings. Strictly speaking we do not measure the pure marshalling overhead, but we chose methods that heavily depend on string arguments to be converted between the internal type in PHARO and the external type expected by the C function.

	Call Time [ms]	Relative Time
NATIVEBOOST	105.29 ± 0.48	1.0
ALIEN	1058.70 ± 4.00	≈ 10.1
C-FFI	282.94 ± 0.48	≈ 2.7
LUAJIT ¹⁰	97.00 ± 1.00	≈ 0.9

Table 4.5: Speed comparison of an `char * getenv(char *name)` FFI call (see Code Listing 4.2).

As a last evaluation of simple C functions with NATIVEBOOST, we call `printf` with a string and two integers as argument. The marshalling overhead is less than for the previous `getenv` example as the strings passed and returned

⁹Downsampled from increased loop size by a factor 100 to guarantee accuracy.

¹⁰Downsampled from increased loop size by a factor 10 to guarantee accuracy.

are smaller. However, `printf` is a more complex C function which requires more time to complete: it has to parse the format string, format the given arguments and pipe the results to standard out. Hence the relative overhead of an FFI call is reduced.

	Call Time [ms]	Relative Time
NATIVEBOOST	371.03 ± 1.02	1.0
ALIEN	1412.37 ± 1.58	≈ 3.8
C-FFI	605.02 ± 0.46	≈ 1.6
LUAJIT	202.40 ± 4.20	≈ 0.6

Table 4.6: Speed comparison of an `int printf(char *name, int num1, int num2)` FFI call

Evaluation. Table 4.3 and Table 4.4 call C functions that return integers for which the conversion overhead is comparably low. However we see that ALIEN compares worse in the case of more complex Strings. Table 4.5 and Table 4.6 show this behavior. For the `getenv` a comparably long string is returned which causes a factor 10 conversion overhead for ALIEN.

4.3.3 Using Complex Structures

To evaluate the impact of marshalling complex types, we measure the execution time for a callout to `cairo_matrix_multiply` described in Listing 4.9. In all cases, the allocation time of the structs is not included in the measurement nor their field assignments. Table 4.7 shows the results.

	Call Time [ms]	Relative Time
NATIVEBOOST	79.00 ± 0.54	1.0
ALIEN	753.82 ± 1.02	≈ 9.5
C-FFI	380.80 ± 5.40	≈ 3.6
LUAJIT	5.66 ± 0.30	≈ 0.07

Table 4.7: Speed comparison of an `cairo_matrix_multiply` FFI call (cf. Listing 4.9)

Evaluation. Table 4.7 shows that NATIVEBOOST outperforms the two other PHARO implementations.

4.3.4 Callbacks

Table 4.8 shows a comparison of `qsort` callouts passing callbacks. Callbacks are usually much slower than callouts.

	Call Time [ms]	Relative Time
NATIVEBOOST	2300.00 ± 2.20	1.0
ALIEN	600.83 ± 0.70	≈ 0.26
C-FFI	NA	NA
LUAJIT	46.13 ± 1.24	≈ 0.02
NATIVEBOOST with Native Callbacks	4.98 ± 0.42	≈ 0.002

Table 4.8: Speed comparison of a `qsort` FFI call (cf. Listing 4.10)

Evaluation. The results show that `NATIVEBOOST` callbacks are currently slower than `ALIEN`'s ones. This is because `ALIEN` relies on specific `VM` support for callbacks making their activation faster (context creation and stack pages integration). On the opposite, `NATIVEBOOST` currently uses small support from the `VM` side and even do part of the work at image side. This `qsort` demonstrates the worst case because it implies a lot of activations of the callback. For each of these calls, `NATIVEBOOST` creates a context and make the `VM` switch to it. To really demonstrate that these context switches are the bottleneck, Table 4.8 also shows the result of doing the same benchmark in `NATIVEBOOST` but using a native callback i.e. containing native code. We do not argue here that callbacks should be implemented in native code but that `NATIVEBOOST` support for callback can be optimized to reach `ALIEN`'s performance at least.

4.4 Implementation Details

The following subsections will first focus on the high-level, language-side aspects of `NATIVEBOOST`, such as native code generation and marshalling. As a second part we describe the low-level interaction of `NATIVEBOOST` with `BENZO`.

4.4.1 Generating Native Code

In `NATIVEBOOST` all code generation happens transparently at language-side. The various examples shown in Section 4.2 show how an FFI callout is defined in a standard method. Upon first activation the `NATIVEBOOST` primitive will fail and by default continues to evaluate the following method body. This is the point where `NATIVEBOOST` generates native code and attaches it

to the compiled method. `NATIVEBOOST` then reflectively resends the original message with the original arguments (for instance `abs:` in the example Figure 4.1). On the second activation, the native code is present and thus the primitive will no fail but run the native code. Section 3.2.2 will gave more internal details about the code activation and triggering of code generation.

Generating Assembler Instructions. Figure 4.2 shows that `NATIVEBOOST` relies on `AsmJit`¹¹, a language-side assembler. `AsmJit` emerged from an existing C++ implementation¹² and currently supports the x86 instruction set. In fact it is even possible to inline custom assembler instructions in `PHARO` when using `NATIVEBOOST`. This way it is possible to meet critical performance requirements. Typically `PHARO` does not excel at algorithmic code since such code does not benefit from dynamic message sends.

Reflective Symbiosis. `NATIVEBOOST` lives in symbiosis with the `PHARO` programming environment. As shown in the examples in Section 4.2 and in more detail in Figure 4.1 `NATIVEBOOST` detects which method arguments correspond to which argument in the `FFI` callout. To achieve this, `NATIVEBOOST` inspects the activation context when generating native code. Through reflective access to the execution context we can retrieve the method's source code and thus the argument names and positions.

Memory Management. `NATIVEBOOST` supports external heap management with explicit allocation and freeing of memory regions. There are interfaces for `allocate` and `free` as well as for `memcpy`:

```
memory := NativeBoost allocate: 4.
bytes := #[1 2 3 4].
"Fill the external memory"
NativeBoost memcpy: bytes to: memory size: 4.

"FFI call to fill the external object"
self fillExternalMemory: memory.

"Copy back bytes from the external object"
NativeBoost memcpy: memory to: bytes size: 4.
NativeBoost free: memory.
```

Code Example 4.13: Example of external heap management in `NATIVEBOOST`

Using the external heap management it is possible to prepare binary blobs and structures for `FFI` calls. In the previous example Code Listing 4.13 the

¹¹<http://smalltalkhub.com/#!/~Pharo/AsmJit/>

¹²<http://code.google.com/p/asmjit/>

`memory` variable holds a wrapper for the static address of the allocated memory. Hence accessing it from low-level code is straight forward. However in certain situations it is required to access a high-level object from assembler. PHARO has a moving garbage collector which means that you can not refer directly to a high-level object by a fixed address.

As explained in more detail in Section 3.2.1 BENZO uses a special array at a known fixed address to deal with this problem. Unlike normal PHARO objects, this array has a known, fixed address that contains pointers to high-level objects. The garbage collector keeps this external roots array up to date. Hence it is possible to statically refer to a PHARO object using a double indirection over the external roots.

4.4.2 Activating Native Code

In this section we present the VM-level interaction of NATIVEBOOST. Even though NATIVEBOOST handles most tasks directly at language-side it requires certain changes on VM level:

- executable memory,
- activation primitives for native code.

Since NATIVEBOOST manages native code at language-side there is no special structure or memory region where native code is stored. Native instructions are appended to compiled methods which reside on the heap. Hence the heap has to be executable in order to jump to the native instructions.

4.5 NATIVEBOOST Limitations

After presenting NATIVEBOOST with all its benefits in detail we also have to shed some light on its limitations in this section. The problems and limitations described in this section were discovered while using NATIVEBOOST extensively in PHARO. Since NATIVEBOOST is based on BENZO for the low-level programming part, most of NATIVEBOOST's limitations are the limitations of BENZO itself that were already presented previously in Section 3.6. In this section we will focus on the high-level problems and leave out the low-level limitations of BENZO as they are not the domain of NATIVEBOOST. As such the major issue with NATIVEBOOST is the lack of a dedicated debugging infrastructure followed by a NATIVEBOOST-specific approach to support platform dependent code.

4.5.1 Difficult Debug Cycles

As a direct consequence from BENZO's shortcomings is the limited debugging support. Once NATIVEBOOST generated the native code for the callout

there is no possibility to interact with the code anymore. We BENZO described in Section 3.6.1 that there is not special debugging mode available and native code is run unprotected. As a result, errors happening inside external libraries have fatal consequences in NATIVEBOOST: the process running the PHARO image is terminated. The core of this problem has to be addressed at BENZO-level and not directly in NATIVEBOOST.

In a future version of NATIVEBOOST, together with improvements of the underlying BENZO debugging infrastructure (see Section 3.6.2), we envision a seamless interaction with the external libraries. There should be no barrier between PHARO and external code, a more sophisticated debugger could dynamically switch context and start displaying more C oriented information in the external library. In the worst case we could still display native instructions and inspect the stack as we step through the external function. In best case we could provide a GDB-like debugging experience with source code and resolved symbol names.

4.5.2 Platform Independence

The second problem we would like to address is platform independence. This is certainly a crucial issue for any framework that deals with native code and as such a main concern of BENZO (see Section 3.6.3). However, the instruction-level architecture support is of secondary importance for NATIVEBOOST as it interacts mostly on operating system level. Rewriting NATIVEBOOST's explicit assembler routines in the platform independent intermediate representation (VIRTUALCPU) presented in Section 3.6.3 would solve the CPU architecture dependency.

Currently NATIVEBOOST is used on three operating system: LINUX, MAC OS X and WINDOWS. Internally NATIVEBOOST already deals with different calling conventions C-functions on the different platforms. Nevertheless, from a user point of view it is mandatory to have a well defined way to deal with platform specific FFI callouts. The current approach is to create a specific subclass for each platform. A simple extension to NATIVEBOOST to allow callout definitions for multiple platforms in a single method would greatly improve this case. The following code example illustrates a possible solution:

FFI

```

unix: [ :builder |
    builder
    nbCall: #(int setenv(
                String name, String value, 1))
    module: NativeBoost CLibrary ];
windows: [ :builder |
    builder
    nbCall: #(int SetEnvironmentVariableA(

```

```

        String name, String value))
    module: #Kernel32
    options: #(optStringOrNull) ].

```

This examples includes the platform specific version for accessing an environment variable. In this case the difference of the two platforms is handled by simply using a different native function. However, already in the case of `getenv` function, `WINDOWS` and `UNIX` implementations behave fundamentally different from `WINDOWS`' `GetEnvironmentVariableA` and a small `PHARO` helper method is necessary to overcome the differences. In this case it would be nice to mix `PHARO` code and `FFI` callouts more vividly and for instance allow inline `PHARO` code in the example above.

4.5.3 Limited Expressiveness

`NATIVEBOOST` has been designed to call a single external function per callout. To our knowledge this is the standard in `FFI` implementations. For most of the use cases this is sufficient but during development we found a peculiar case, when using `fork`, where two C function consecutive calls have to be made. `fork` creates a new process at `os-level`. It returns 0 in the newly forked process and the new process id to the original parent process. Hence a typical `fork` usage in C looks as follows:

```

1 pid_t pid;
2 int status = 0;
  // spawn child process
4 if(!(pid = fork())) {
      // execute code in the child process
6      ....
      // stop the child process
8      exit(EXIT_SUCCESS);
  }
10 // in the parent process wait for the child process
    waitpid(pid, &status, 0);

```

Currently it is not possible to express this directly in `NATIVEBOOST` due to the `vm`. Imagine the following hypothetical `NATIVEBOOST`-based implementation of the previous example:

```

pid := FFI fork.
pid isZero
  ifTrue: [
    self childProcessMethod.
    FFI exit: 0 ]
  ifFalse: [
    FFI waitPid: pid for: 10 seconds ]

```

The first two lines pose a significant problem, still ignoring the implementation details of the rest of the method. What happens when we call `fork` with

an FFI callout? Essentially this creates a fork of the whole VM running PHARO which has some side-effects since the resources between the child and the parent VM process are shared. For instances the VM has a separate process running for handling input events, which does not get forked automatically. Additionally we see certain file handles that are not properly recreated for the child process. In consequence when the FFI callout for `fork` finishes, the VM continues interpreting PHARO code which in some cases stops to work due to the aforementioned side-effects. This could technically be avoided if the solve purpose of the child process is to execute some native code in the background and the exit. One would have to make sure that the sequence, `fork`, `PID` check, `exit` happens all in native code without handing over execution to PHARO in between. Which brings us back to the original observation that we cannot create a combined FFI callout for multiple methods in NATIVEBOOST.

To solve this we could allow NATIVEBOOST to mix assembler instructions (or the more abstract VIRTUALCPU instructions) with multiple callouts. Currently this almost possible by manually invoking the callout generator for different C functions. However, there are certain side-effects with the stack-management which require more work.

4.5.4 Startup Recursion

Starting with PHARO 3.0 we tried to slowly replace VM plugin functionality with direct FFI callouts at language-side. This is an attempt to make language more open and shift a part of the development done in C at VM-level to PHARO. The driving motivation is the same as for NATIVEBOOST itself: accessible and flexible code.

The OS environment implementation based on NATIVEBOOST described earlier was such an attempt. In return it allowed us to add functionality to access to known directory locations under LINUX by directly or indirectly querying environment variables such as HOME. In a second refactoring the functionality for opening the changes-file in PHARO containing the changelog was made more flexible supporting more file locations. This last change introduced a recursive dependency that is not visible on the first sight and illustrated in the following figure. Essentially this meant that in certain cases where NATIVEBOOST required to compile the native code for callout it was impossible to start up PHARO. Once the native code was cached this recursion chain was broken and subsequently PHARO started up well.

This particular issue was solved in PHARO by caching the argument names in NATIVEBOOST used for assigning the PHARO arguments to the callout parameters (see Section 4.2.2). The downside of this approach is that NATIVEBOOST reimplements part of PHARO's existing reflection. Of course there are

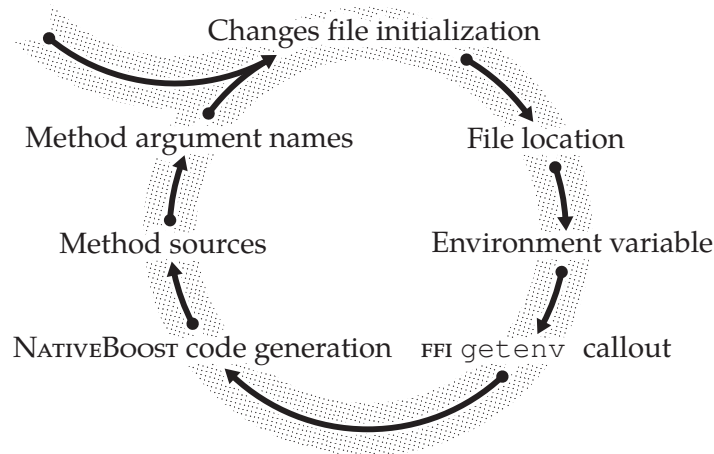


Figure 4.4: When using NATIVEBOOST in critical PHARO components used during the startup we run into a infinite recursion due to cyclic dependencies.

other possibilities to avoid the described situation: deferred startup logics, lazy startup or storing the complete PHARO source code in the image. However, they would leave the general problem of using NATIVEBOOST during startup which will happen more often if more VM plugins are replaced with FFI callouts. And even more general, how do we enable NATIVEBOOST on a system that does not allow dynamic code generation? The only solution to this problem is to generate the FFI callouts upfront and make these binaries available to the final product. In this case NATIVEBOOST would need to modes: an interactive development mode and a static deployment mode. The development mode is how we describe NATIVEBOOST so far, programmers can interactively create callouts. In the static or deployment mode all the dynamic callouts written using NATIVEBOOST are replaced by PHARO plugin primitives. Only that the plugins are generated from the original NATIVEBOOST callouts rather than compiled from standard C sources. At the current state of the FFI

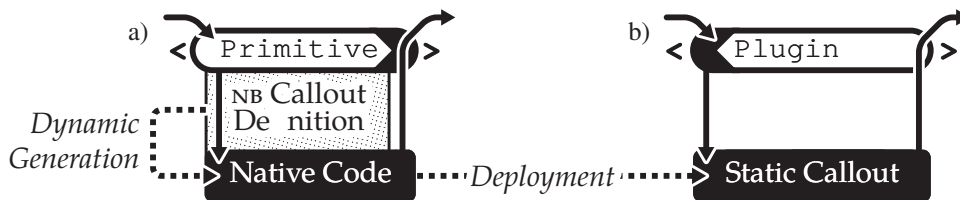


Figure 4.5: Comparison of the callout activation for a) the development mode where native code is created on demand and b) the deployment mode where we use pre-generated native code.

library and the underlying assembler from BENZO this is only feasible for the x86 architecture while for instance ARM support is missing. Next to the additional assembler backends we also have to implement the platform specific

binary formats such as `ELF` under `LINUX` or `MACH-O` under `OS X`. This would work much similar to the implementation used in the research `SMALLTALK` implementation `PINOCCHIO` [50].

4.5.5 Performance

Even though `NATIVEBOOST` shows good overall performance when it comes to callbacks it does not keep up with other `SMALLTALK`-based solutions. In the current development phase not much attention was paid to callback performance as it is not a common use case for `FFI` callouts. Fast callbacks require close interaction and specific modifications at `VM`-level. However, initially `NATIVEBOOST` kept the modifications to the `VM` at a minimum. We assume that we can reach the same performance as `ALIEN` relying on the same low-level implementation.

As a second issue we would like to address the callout overhead by using an already existing `JIT` integration of `NATIVEBOOST`. Currently the `VM` has to leave from `JIT`-mode to standard interpretation mode when it activates an `NATIVEBOOST` method. This context switch introduces an unnecessary overhead for an `FFI` callout. A current prototype directly inlines the native code of a `NATIVEBOOST` method in the `JIT`. Hence the cost for the context switch plus the cost of activating the `NATIVEBOOST` callout primitive can be avoided.

4.6 Conclusion

In this chapter we presented `NATIVEBOOST` a novel approach to foreign function interfaces (`FFI`). Unlike other implementations `NATIVEBOOST` does not rely on specific plugins for doing `FFI` callout. Instead it is implemented on top of the generic low-level programming framework `BENZO` which we presented earlier in Chapter 3. As such `NATIVEBOOST` is implemented completely language-side.

`NATIVEBOOST` Performance. Using a in depth evaluation of `NATIVEBOOST` comparing against two other `SMALLTALK` `FFI` implementations and `LUAJIT` we showed in Section 4.3 that our language-side approach is competitive. `NATIVEBOOST` reduces the callout overhead by more than a factor two compared to the two closest `SMALLTALK` solutions.

Compared to `LUAJIT` there is still space for improvements. We measured a factor 30 lower calling overhead due to a close `JIT` integration. However for typical `FFI` calls the absolute time difference between `NATIVEBOOST` and `LUA` is roughly 30%. With a partial solution ready to integrate `NATIVEBOOST` closer with the `JIT` we expect to come closer to `LUA`'s performance. We implemented

already a JIT enabled `BENZO` primitive that inlines the language-side native code into the jitted code and thus avoids the overhead of the activating a primitive each time. However, this requires certain changes when generating native code in `NATIVEBOOST`. Namely, we have to respect the register strategy used by the JIT. Due to these non-trivial changes we have not ported `NATIVEBOOST` to the JIT-enabled `BENZO` primitive yet.

Furthermore we showed that `NATIVEBOOST` essentially combines VM-level performance with language-side flexibility when it comes to marshal complex types. New structures are defined practically at language-side and conversion optimizations are added transparently.

`NATIVEBOOST` Limitations. Much of the limitations of `NATIVEBOOST` are due to shortcomings of the underlying `BENZO` framework. The most prominent effects are the ones already mentioned earlier in Section 3.6 and concern the debugging cycle. Currently the great performance and flexibility comes at the price of hard crashes when writing faulty code. In support of `NATIVEBOOST` callouts we state that C code is equally prone to these bugs. A possible solution would be a debugger that is also capable of stepping through native code.

`NATIVEBOOST` Outlook. We have shown that `NATIVEBOOST` is a stable and fast FFI framework and thus a clear validation of the underlying `BENZO` framework. Much like `BENZO`, `NATIVEBOOST` requires still some work to improve the debugging cycle to provide a barrier free development experience for `PHARO` programmers. However, we believe that with this language-side approach changes to the FFI framework are much simpler to perform and thus eventually will encourage more contributors to improve `NATIVEBOOST`.

After presenting the validation of a real-world application of the `BENZO` framework in this chapter we will now focus on two research applications in the following chapter.

BENZO PROTOTYPE APPLICATION VALIDATION

Contents

5.1	WATERFALL: Dynamic Primitives	82
5.2	NABUJITO: Language-side JIT Prototype	93
5.3	BENZO Applications: Outlook and Summary	106

Introduction

In Chapter 4 we presented `NATIVEBOOST`, a mature language-side `FFI` implementation that makes heavy use of `BENZO`'s infrastructure. `NATIVEBOOST` is only one of three applications that are based on `BENZO` that were initially outlined in Section 3.3. While `NATIVEBOOST` is considered stable, the two other applications are currently only prototypes: dynamic primitives and a language-side `JIT`. Hence we will present the two solutions combined in this chapter.

As first we will present `WATERFALL`, dynamic primitives based on `BENZO`. `WATERFALL` takes advantage of the metacircular approach of `PHARO`'s `VM` and makes the primitive definition available at runtime. This is a step forward from the typical metacircular approach where the whole reflective power of the host environment can only be used at compile-time. Once the `VM` is compiled, all the high-level definitions that existed at compilation time are no longer accessible from language-side. `WATERFALL` tries to make a fraction of the original compile-time definitions accessible.

The second prototype, `NABUJITO` a language-side `JIT` compiler takes the core idea of `WATERFALL` even further. `WATERFALL` is capable of defining new primitives at runtime which are not reentrant: it is not possible to activate `PHARO` methods from within primitives. However, this is what happens in jitted methods: it is possible to switch seamlessly between native methods and standard `PHARO` methods using bytecode evaluation. Much like the primitives, the `JIT` can not be changed from language-side and this is where we bring `NABUJITO` into play. `NABUJITO` reimplements the `VM`-level `JIT` compiler at language-side and uses `BENZO` to install the native code.

5.1 WATERFALL: Dynamic Primitives

In this section we present *WATERFALL*, a compiler toolchain developed by Guido Chari that allows primitives to be changed dynamically from language-side. *WATERFALL* in the core is a *SLANG* compiler implemented purely in *PHARO* relying on the existing *BENZO* infrastructure. In a separate work [16] we show how *WATERFALL* is capable of recompiling whole *VM* plugins.

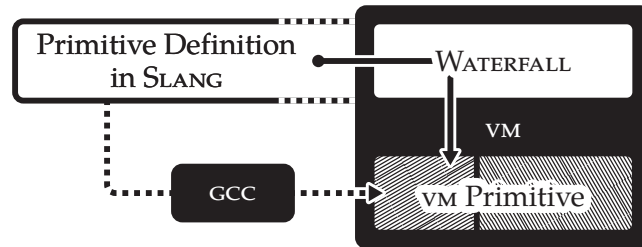


Figure 5.1: *WATERFALL* uses the *SLANG* definition of the *VM* to generate new primitives on the fly, creating an incremental version of the default *VM* compilation process that compiles the whole *VM* using a C compiler (*gcc*).

5.1.1 Background

WATERFALL is our second application on top of *BENZO* after *NATIVEBOOST*, the complete *FFI* library previously presented in Chapter 4. *NATIVEBOOST* uses *BENZO* to generate the glue code between *PHARO* and the external library. Even though *NATIVEBOOST* is extendable it is not used to directly synthesize new functionality, the main functionality is defined the external libraries typically written in a low-level language such as C. Interestingly, the *NATIVEBOOST* methods containing the callouts behave almost like the existing primitive methods of *PHARO*. These primitives define a hook into *VM*-level native functionality. In *PHARO* the same mechanism is also used to activate plugins which are again similar to an *FFI* callout from language-side. However, primitives and plugins are statically defined and modifications happen outside *PHARO*. This is where the domain of *WATERFALL* begins.

WATERFALL provides infrastructure to dynamically compile and install primitives on top of the *BENZO* infrastructure at language-side in *PHARO*. As we will describe in more detail in the following sections, the *PHARO* *VM* is written in a metacircular fashion. Hence the definition of plugins and primitives can be loaded in standard *PHARO*. Typically this happens only at compile-time of the *VM*, where these definitions are exported to C and compiled to the *VM* binary. Once compiled, the original high-level description of primitives and plugins is no longer accessible from *PHARO*. As a consequence, existing primitives or plugins can not be changed at runtime.

WATERFALL brings the static primitive definitions to live again. Just loading the original definitions in PHARO does not bring them back to live, even though we can now inspect the definition and browse the sources. WATERFALL compiles these definitions to native code and installs them with BENZO as new primitives. With this infrastructure primitive and plugin modifications are not limited to VM compilation time.

A Metacircular VM Written in SLANG. WATERFALL is implemented in PHARO which uses the COG VM¹, originating from the SQUEAK VM [28]. The VM itself is written in a dialect of SMALLTALK called SLANG that is essentially limited to the functionality that can be expressed with standard C code. SLANG serves for two purposes: a high-level C preprocessor, a interactive simulator of the VM. The first point has severe consequences. 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. This fact is also visible in the simulator for the VM. If SLANG were SMALLTALK, separate parts of the VM could be directly evaluated. However, since SLANG is bound to C expressions, the simulator sets up a byte array as memory. The simulated VM then accesses this byte array as if it were the native memory.

In conclusion we see that the PHARO VM has an abstract representation of the VM available for simulation. This abstract representation is then used to generate C sources, already lowering the abstraction level. After compiling the C sources the original representation of the VM is not directly accessible anymore. For instance, even debug symbols are usually stripped from the final binary for performance reasons. Of course this implies that the VM can not be changed nor directly inspected from language-side.

Primitives in PHARO. PHARO is a highly reflective environment where classes and methods can be changed at runtime, even the current execution context is accessible. For instance this is used to implement an exception mechanism purely at language-side in PHARO. However, some features can not be implemented at language-side. PHARO uses primitive methods, that instead of evaluating PHARO-code switch to a VM routine. As already partially explained in Section 3.2.1, whenever a method is compiled with the `primitive` pragma as shown a flag is set on the `CompiledMethod`. If the VM tries to activate such a method, instead of interpreting the bytecodes it calls the corresponding function at VM-level [27]. We distinguish three categories of primitives based on their functionality: certain parts of the language semantics, OS-level

¹<http://www.mirandabanda.org/cogblog/>

functionality that can not be implemented in PHARO itself and a third less important category where performance is critical.

As we mentioned in the previous paragraph, these primitives are bound to the VM and can not be changed at runtime. However, for a certain subset of these primitives we can write language-side substitutes in pure PHARO-code. These primitives are called non-essential and are mainly used for optimization purposes. In contrast there are essential primitives which are for instance used during start up of the PHARO environment. Two prominent examples of essential primitives are the ones used for creating new objects or activating a block.

Instrumenting Primitives. In the context of WATERFALL we are interested in which parts of the system we can modify and thus we draw our attention to these essential primitives. The only way to modify these primitives is by creating wrappers but that brings a new problem. Imagine that we wrap around the primitive which creates a new object. What happens now if the additional wrapper code needs a new object? It will call the very same primitive that we just wrapped, without protection this causes infinite recursion. Since technically the wrapper code should live at a different abstraction level than the original primitive we have found ourselves mixing meta-levels [17].

The most radical approach to avoid this meta-recursion is to change the primitive externally. In the case of PHARO this means changing the SLANG sources, exporting and compiling the primitive and restarting the PHARO environment on top of this changed VM. However, this approach stands in contrast to the reflective nature of PHARO where most functionality can be changed at runtime. Also it is not always suitable to restart the PHARO process to modify a small part of the system.

5.1.2 WATERFALL'S Contribution

Following the implementation overview of the PHARO VM and the differentiation of different primitives we identify two main benefits of changing VM primitives at runtime with WATERFALL:

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

Reducing VM Complexity. Low-level VM extensions are only justified in the presence of strong performance requirements (see Section 3.5). All non-essential primitives fall into this category since these primitives can be implemented in PHARO without restrictions. However, in certain cases for

performance a language-side implementation is unsuitable. Additionally we already know that these primitives are available as `SLANG` code at `VM` generation time. Using `WATERFALL`, these primitives can be implemented at language-side based on the unmodified `SLANG` sources. This means that these primitives become first-class citizens of the high-level environment and thus evolve with less effort. Thus, `WATERFALL` opens new possibilities of changing `PHARO` that were previously possible only with significant overhead.

Essential Primitives. For essential primitives the previous argument does not hold since a static version is needed for a correct startup of the system. These primitives can not be directly replaced by a language-side implementation using `WATERFALL`. Even though `WATERFALL` itself avoids meta-recursion by generating low-level code with `BENZO`. However, `BENZO` itself relies on essential primitives as it is written in `PHARO`. This imposes certain restrictions how and when these essential primitives can be modified with `WATERFALL` during system startup. These restrictions are more related to the underlying `BENZO` infrastructure than `WATERFALL`. For instance already exposed similar limitations with the `BENZO`-based `FFI` when used during startup (see Section 4.5.4). Nevertheless, nothing prevents from replacing essential primitives at runtime with customized versions, once the system startup is completed.

Extended Primitive Instrumentation. Instrumentation of essential primitives from language-side is an error-prone task falling in many cases in non-termination due to previously described meta-recursion. An example of this behavior, can be observed when changing the essential `basicNew` primitive, which is responsible for instantiating new objects. Only very limited instrumentation is possible at language-side, for instance counting how many instances have been created. This only works since the `VM` internally does not represent small integers as full objects. However, this is only true up to some extent. Small integers bigger than 2^{30} are transformed to a more expensive object representation since they no longer fit in a machine word of the 32-bit `VM`. These big integers will use the `basicNew` primitive again as they are not implemented in the `VM` but in at language-side. Thus, we are back the original problem of running into meta-recursion. So even this very simple example has unwanted side-effects that are not directly visible. More complex instructions tasks will inevitably suffer from the same problems.

Using reflective techniques it is possible to escape from this meta-recursion, however, with a considerable overhead. `WATERFALL` avoids these issues since the instrumentation code for primitives will be implemented at the lowest level on top of `BENZO`. In Section 5.1.4 we show how `WATERFALL`,

the BENZO based approach for generating primitives on the fly, outperforms the reflective solutions for primitives instrumentation.

5.1.3 WATERFALL Implementation

WATERFALL uses BENZO's mechanism for replacing primitive methods with customized versions that are nativized dynamically as described in Chapter 3. The loophole described there is exploited by WATERFALL to enable dynamic modification of VM behavior and hence bring primitives to life at language-side. From a high-level point of view WATERFALL provides two services which work transparently:

1. Compilation of SLANG code on demand (lazily).
2. A clear interface for executing, at runtime and from language-side, the native code generated.

The first item allows to change the code of primitives at language-side and generate the corresponding native code when needed. It also provides the possibility to write methods or functionality with the same SMALLTALK syntax but with a static semantic. It consists essentially of a transformation toolchain that transforms the SLANG sources to native code using a BENZO-based compilation toolchain.

The second item enables the execution of the dynamically generated native code. This includes for instance the finding of addresses of VM internal symbols and all the effort to link the two worlds, SMALLTALK and native. WATERFALL relies on BENZO for most of this low-level functionality. In particular NATIVEBOOST, the BENZO-based FFI presented in Chapter 4, is used for interfacing with C libraries (`dl sym`).

Architecture Overview. The WATERFALL infrastructure is mainly divided in the following two parts:

- the installed SLANG sources,
- a BENZO-based compilation toolchain.

The WATERFALL compiler transforms the SLANG sources to native code through various transformation steps as show in Figure 5.2. In order to work properly WATERFALL needs the complete SLANG sources for compilation unit (primitive or plugin) to be loaded upfront. Once loaded in the PHARO image the AST of the SLANG sources are available which form the input for the WATERFALL compiler. This means that it is possible to write custom plugins in PHARO and transform them using WATERFALL as long as the written PHARO code uses the restricted SLANG subset. As mentioned earlier, the major difference to normal

PHARO code is the lack of real polymorphism since SLANG is more like C with a SMALLTALK syntax.

Technically the WATERFALL compiler takes over the part of the SLANG to C converter and of GCC in the normal VM compilation process. WATERFALL, much like the SLANG to C converter, has to take care of certain type information present in the SLANG sources. For instance we extract from the type information if arguments are used by value or reference. With this information we generate native code using a simple stack based strategy for temporary variables. As for the part of GCC, WATERFALL in its current state is of course far less complex and the resulting the native code is inferior to GCC's optimized output. To simplify the prototype WATERFALL only uses a simple stack strategy instead of register allocation for temporaries. Additionally WATERFALL does not use intermediate representations (IR) such as static single assignment (SSA) to perform elaborate optimizations [5, Ch. 1].

Compilation Steps. As shown in Figure 5.2 the WATERFALL compiler transforms the AST of the SLANG input to PHARO primitives.

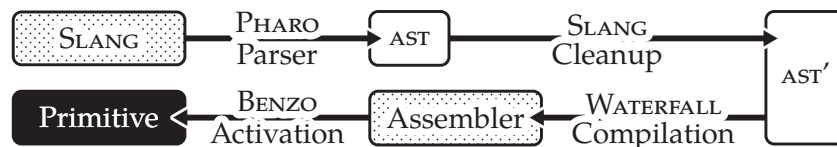


Figure 5.2: WATERFALL loads the SLANG sources and lowers the AST to assembler instructions. Using BENZO the binary code for a primitive is activated and installed.

We divide the WATERFALL compiler into four distinct steps:

SLANG to AST: The first step is to access the AST of the SLANG source method which happens automatically by loading the SLANG code in the PHARO image. At this stage WATERFALL also recursively collects the set of reachable SLANG methods.

AST Purification: In a second step certain expressions of the original SLANG AST are transformed into custom WATERFALL expressions that can be easier transformed later on. For instance WATERFALL converts C macros that are supported in SLANG which of course only make sense when using a standard C compiler.

AST to ASM: The real native compilation happens in the third step where an AST-visitor creates assembler instructions using BENZO's ASMJIT. At this point external symbols are statically resolved and directly inlined in the final ASM code.

ASM to Primitive Although not strictly part of the compilation, in the fourth step the final native instructions are installed as a primitive methods using BENZO (see Section 3.2 for more details).

Dynamically Replacing Primitives. After explaining the general architecture and the different compilation steps of WATERFALL we shed some light on how the primitives are actually installed. In reality we rely 100% on BENZO for this feature. Once the native code is generated we transform the target method to a special BENZO-enabled method that contains the native code. This procedure is explained in detail in Section 3.2 where we show the implementation details of BENZO.

From a user point-of-view we only have to make sure that the corresponding SLANG sources are available and then hand over that source method to WATERFALL to compile and install it. Once the installation is complete, the resulting BENZO-enabled method will contain behave like a SLANG primitive compiled with the original approach using GCC.

Dynamically Replacing Plugins. In PHARO there is no real distinction between primitives and plugins as we illustrate with the following code snippets. The first one depicts an essential primitive to allocate new objects. The second code example shows the a plugin primitive to open a new file stream.

```
basicNew
  <primitive: 70>
  OutOfMemory signal.
```

Code Example 5.1: Object»#basicNew Primitive

```
open: pathString writable: writableFlag
  "Open a file at the given pathString, and return
  the file ID obtained."
  <primitive: 'primitiveFileOpen' module: 'FilePlugin'>
  ^ nil
```

Code Example 5.2: FilePlugin»#open:writable: Plugin Primitive

The main difference between primitives and plugins is only how they are distributed. Primitives are inlined in the VM and can not be loaded at runtime, while plugins can be loaded dynamically and are bundled separately. That also means that there is no difference in handling plugins for WATERFALL, the compilation and installation process is exactly the same.

5.1.4 WATERFALL Validation

After explaining the implementation details of WATERFALL we would like to present a thorough evaluation of the WATERFALL infrastructure. We split up

the validation in two parts following the outlined applications of WATERFALL in the introduction. The first part describes the performance of WATERFALL when used for instrumenting primitives. This is the major field of application for WATERFALL as it stresses its dynamic nature. In contrast to that we evaluate the performance of a WATERFALL compiled plugin in the second part of the validation. Evaluating a whole plugin puts more stress on the quality of the generated code than the fact that we can dynamically modify primitives. A more detailed analysis of WATERFALL is also available separately [16].

Validation of Dynamic Primitives

In this first part of the WATERFALL validation we compare the performance of WATERFALL generated primitives in PHARO. In the first part we simply measure the speed of a dynamically replaced primitive, while in the second we add instrumentation overhead. For the simple replacement we choose the simple integer operation "greater than" (>) and for instrumentation the more complex `basicNew` primitive.

Simple Dynamic Primitives. In this first validation we compare the speed of the WATERFALL generated code on a simple "greater than" primitive. The primitive is rather simple as it only works on small integers arguments and delegates the functionality for other types to its superclass. The code for the `SmallInteger` operation looks as follows.

```
> aNumber
  <primitive: 4>
  ^super > aNumber
```

The fallback code at the end of the method triggers a slower "greater than" implementation on the super class `Integer` which mostly deals with the multitude of possible arguments to `>`.

```
> aNumber
  aNumber isInteger
    ifFalse: [
      ^ aNumber
        adaptToInteger: self andCompare: #> ]
  self negative == aNumber negative
    ifFalse: [ ^ aNumber negative ].
  self negative
    ifTrue: [ ^( self digitCompare: aNumber) < 0 ]
    ifFalse: [ ^( self digitCompare: aNumber) > 0 ].
```

For comparing performance of the "greater than" primitive we use three different approaches:

1. the standard primitive provided by the vm,

2. the fallback language-side implementation that is triggered whenever the standard primitive failed,
3. the reimplementation with WATERFALL (not instrumented).

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 5.1.

	Running Time [ms]	Relative Time
Unmodified	6.4 ± 2.8	1.0
Fallback	195.0 ± 3.2	≈ 30.0
WATERFALL	22.8 ± 3.4	≈ 3.6

Table 5.1: 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.

Essential Primitive Instrumentation. As a second validation target for primitives we chose to instrument `basicNew` which is a critical primitive for object allocation. Like the previous "greater than" primitive this belongs to the set of essential primitives that are used during startup of the image. For instrumentation `basicNew` is again a rather tricky target as wrong code easily leads to infinite recursion. However, this can be avoided with a rather costly recursion guard. We chose a rather simple instrumentation method by simply printing the address of the allocated object to the standard output stream. We validate the four flavors of the `basicNew` primitive:

1. the unmodified primitive,
2. a reflectively instrumented primitive with a recursion guard written in PHARO,
3. a WATERFALL generated and instrumented version,
4. a WATERFALL generated version without instrumentation.

We measure again with the same setup as for the previous validation of the "greater than" primitive. The outcome of this validation is shown in Table 5.2. Again the results present a similar picture as for the "greater than" validation.

	Time [ms]	Relative Time
Unmodified	0.28 ± 0.32	1
Secure reflective instrumentation	27.72 ± 0.80	≈ 99
WATERFALL-based instrumentation	7.72 ± 0.54	≈ 28
WATERFALL-based non-instrumentation	7.08 ± 0.46	≈ 25

Table 5.2: Slowdown comparison for instrumentation of the essential primitive `basicNew`.

However, since we added instrumentation this time, the reflective PHARO is significantly slower than the unmodified version of the primitive. This proves our theory that in certain performance critical cases reflective solutions are not sufficient. While we were able to circumvent the recursion problem rather elegantly, the recursion guard is simply too slow to be used by default. Compared to that, the WATERFALL-based instrumentation is a factor 3 faster than the reflective solution. We see that the instrumentation overhead compared to the non-instrumented WATERFALL version is in the range of only 0.7ms whereas in the PHARO version the overhead is several magnitudes higher. Unlike for the simpler "greater than" primitive WATERFALL is slower: factor 25 instead of only a factor 3.6 previously. This shows that there is certainly room for performance improvements for WATERFALL.

5.1.5 WATERFALL Limitations and Outlook

WATERFALL is still a research prototype and thus there are several issues that problems that require attention with the most obvious one being performance. We have shown that WATERFALL is fast enough to compete against dynamic primitive instrumentation written at language-side, but when compared to native solutions we are still up to two magnitudes slower. For simplicity WATERFALL currently does not apply any optimizations which still leaves room for improvements. For instance we do not apply register allocation yet. However, in our eyes it does not make sense to implement a specific register allocator for WATERFALL itself. Instead, we envision to use a future platform independent intermediate representation of BENZO that we presented in Section 3.6.3. This way most optimizations only require one implementation from which all BENZO applications benefit. Using this new IR would have very little impact on the current WATERFALL compiler infrastructure as we would only have to replace the AST to ASM compilation step. Instead of generating the ASM we use the BENZO IR and let BENZO generate the native code for the primitives.

WATERFALL is currently only a research prototype that is not used in pro-

duction. Also we have seen that there is a significant overlap with the `NATIVEBOOST_FFI`. For example, many plugins wrap around existing external libraries and thus are perfect candidates for `NATIVEBOOST`. Even though `WATERFALL` would add a lot of flexibility for such plugins, we believe that `NATIVEBOOST` is more intention revealing and less confusing than dealing with the semantic differences of `SLANG` code over `PHARO` code. Nevertheless, this still leaves the big field of instrumentation open for `WATERFALL`. Additionally, for documentation purposes it makes sense to load the `SLANG` definition of all the essential primitives into the `PHARO` image. In this case `WATERFALL` would be a perfect way to bring these primitives to live for exploration purposes.

5.2 NABUJITO: Language-side JIT Prototype

In this section we present NABUJITO, a BENZO-based approach for a language-side JIT compiler that tries to replicated the behavior of the existing JIT as shown in Figure 5.3.

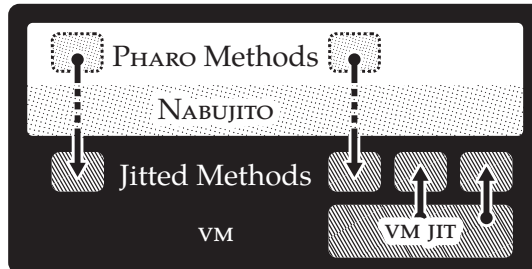


Figure 5.3: NABUJITO tries to replicate the JIT which resides in the VM. We use BENZO to compile native versions of PHARO methods that match the results generated from the existing JIT.

We have presented two other BENZO applications so far: a FFI and previously the dynamic primitives. Both of these applications have in common that the switch or call to native code happens at the end of the PHARO stack as shown in Figure 5.4. We see that the BENZO primitive marks the enter and

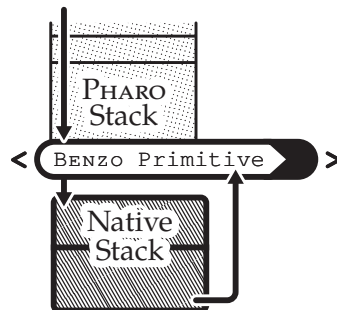


Figure 5.4: WATERFALL and NATIVEBOOST use native code at the end of the PHARO stack.

exit point for native code. As a result the native is isolated from the rest of the stack. However, this is no longer the case if we use BENZO to generate native code for PHARO methods like the VM-level JIT. Figure ?? shows how the BENZO-based native code is embedded into the PHARO stack. Hence a BENZO-based JIT has to closely interact with the existing infrastructure since it is taking control over a part of the interpretation itself. For instance it is not possible to just generate native code following a different stack usage than the internal JIT. If we would do so we would inevitably create a conflict. We will thus present first the details of the existing JIT used by the PHARO VM to shed light on the difficult implementation of NABUJITO.

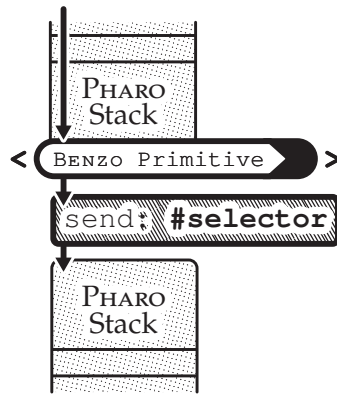


Figure 5.5: NABUJITO generates native code for PHARO methods which can themselves activate other methods. Hence, the BENZO lies in between PHARO contexts.

5.2.1 Background

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.

The JIT of the PHARO VM. The PHARO VM (Cog) 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 [7]. 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 [37, 38]. The existing JIT infrastructure is implemented in SLANG [9, Ch. 5] as the rest of the VM.

To understand the upcoming implementation issues of NABUJITO we have to dive into the details of PHARO'S JIT. PHARO uses a flavor of the COG VM which evolved in several steps from a simple bytecode interpreter. A successful and fast JIT implies a VM that uses the native stack.

The original SMALLTALK-80 blue book implementation foresees a spaghetti-stack where all contexts are normal objects on the heap. This design simplifies the VM implementation significantly since there is no special treatment necessary for blocks. Also this makes it rather easy to implement PHARO'S feature to access the current context using the special `thisContext` variable.

However, the down side of this implementation is that it puts stress on the gc and prevents more aggressive optimizations on the stack. For each message send a new context has to be allocated and on each return contexts have to be reclaimed. It would naturally be more efficient to use the native stack which allows for cheap allocation and precise reclaiming of method context. While this mapping can be done rather easily there are three properties of PHARO that make this hard: blocks, non-local returns and the mentioned `thisContext`. Eliot Miranda eventually succeeded to implement an efficient mapping scheme for the COG VM that is based on the original work done by Peter Deutsch and Allan Schiffman [23].

Even so the basic concepts of the native stack mapping are easy to understand the final implementation is tricky details. Real closures that outlive their outer method activation context make the mapping difficult. At the same time all the reflective capabilities to modified the stack from within PHARO have to be supported. This, in return, limits the optimization opportunities. COG chose a path in between where most reflective modifications of the stack are permitted. However, in certain exotic edge cases the VM does not support the operation.

After supporting the native stack the next optimization in line is the real JIT infrastructure where the VM generates native code on the fly. In COG there is a bytecode compiler that generates a simple intermediate representation which then is used to generate the final native instructions. The IR makes it easier to support new platforms next to the default 32-bit x86 implementation. COG applies minor optimizations like a simple register allocation strategy to lower the stress on stack usage. The most underestimated optimization is the fact that all the native code for the jitted methods is stored in a compact separate memory region. This lowers the chances of cache misses, an ever growing problem on modern CPU architectures.

Figure 5.6 gives an overview of the memory separation used by COG. New objects are allocated in the young space which uses a fast semispac gc



Figure 5.6: COG Memory Model Overview: Fixed-sized JIT space, slow changing old space and fast young space.

with frequent reclaiming. Objects that survive a gc pass move to the old space where infrequent reclaims happen. Separated from the two memory regions where normal PHARO objects reside is the JIT space dedicated for native code. In COG the JIT space has its own gc strategy tailored to native code which

is stored in a structure called `CogMethod`. Each jitted `PHARO` method has a corresponding `CogMethod` with native code which resides in the `JIT` space. The `CogMethod` caches certain information such as the selector or number of arguments. Again this improves code locality as all the frequently accessed information resides in the `JIT` space. Figure 5.7 gives an overview of the `CogMethod`. We see that additionally to the cached meta information there is relocation information (method map) attached to the `CogMethod`. This is used to update object reference, typically to selectors, from the native code in sync with the objects that were moved in a `GC` pass. The same information is also used to update jumps to other native code in the `JIT` space if the dedicated `JIT GC` performs a compaction.



Figure 5.7: Cog Method: Compiled method representation at `JIT`-level residing in the `JIT` memory space.

So far we explained how `Cog` uses native stack mapping for performance reasons and how the basic `JIT` compiler works. To limit the stress on the CPU cache only the most used methods are jitted. `Cog` uses a hierarchy of inline caches to avoid the costly method lookup and checks if a method is already jitted or not. Message sends from one jitted method to another hence happen with a very low overhead. However, due to the limited size of the `JIT` space that still means that infrequently used methods are evaluated using the existing bytecode interpreter. Since `Cog` uses stack mapping this means that the C-based bytecode interpreter runs on the same native stack with two slightly different strategies. For instance, typically the C stack depth is limited to a predefined constant, whereas the `PHARO` stack can grow as big as the whole heap. Which means that recursion in a `PHARO` program is not a direct danger. In the bytecode interpreter the language-side recursion is not directly present, since the interpreter only fetches and evaluates bytecodes in a loop. Thus, switching from `JIT`-mode to bytecode interpretation requires some additional work to avoid negative side-effects. Essentially, `Cog` separates the native stack for the `JIT` and the bytecode interpreter. Each time `Cog` has to switch execution mode it goes through a trampoline routine. The trampoline will exchange the two native stacks and jump to the proper location to continue the execution of the bytecode interpreter from the native `JIT` mode

or vice versa. If COG would simply call back to the C interpreter a new stack frame would be allocated, notably on the existing native JIT stack. This stack frame would persist as the bytecode interpreter continues running normal PHARO message sends.

Limitations of VM-level JIT Compilers. In the context of NABUJITO we split the JIT infrastructure into separate parts. The major part is to have a VM that uses stack-mapping. In the case of a bytecode-based interpreter, we assume that the VM provides routines to switch between a bytecode interpretation context and a low-level native execution context. With NABUJITO we move the JIT compiler, the part that generates native code at runtime, from the VM to the image. Since the JIT compiler 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 at VM-level.
- The JIT reimplements primitives for optimization reasons resulting in code duplication.

Optimization Limitations for PHARO. In SMALLTALK methods tend to be very small and it is considered good practice to delegate behavior to other objects. This implies that several common optimization techniques for static languages do not work well. Dynamic method activation does 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 PHARO, compiling a method to bytecode 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 guarantees that the VM stays as long as possible in the JIT context (see Section 3.2.1 on page 36). Additionally this enables new performance optimizations that for instance are hard to achieve with standard compliant C code. A typical example is the integer addition which has to deal with overflow checks and conversion of tagged integers. In Section 5.1 we describe how WATERFALL suffers a similar constraint. WATERFALL manually

defines such primitives in terms of native assembler instructions through the language-side BENZO interface. NABUJITO reuses the same optimized primitives so we rely on a single optimized definition which is shared among all native code libraries.

5.2.2 NABUJITO Implementation

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 existing intermediate bytecode representation. Additionally we reimplemented vital native routines for the JIT which are not directly exported by the VM using BENZO. Nabujito relies on the following VM-level infrastructure to manage and run native code for any PHARO method:

- native stack management,
- routines for switching contexts,
- JIT-level memory management for code segments.

The native stack mapping is an implicit requirement for an efficient JIT. Since this feature requires deep changes at VM-level we can not alter or reimplement this at language-side. However, the routines for switching between JIT and non-JIT execution context can be mostly reimplemented at language-side. We only chose to implement a small subset of them with BENZO that were directly required for performing message sends. Some of the helper routines' C-level addresses are easily accessible from language-side using `dlSym`. Hence we reuse these for simplicity and only reimplemented the ones that are "hidden". The last item we reuse, JIT-level memory management, poses certain problems as we have little to no control over this from language-side. There is no well-defined interface to interact with the JIT from language-side in PHARO. However, to properly interact with the JIT we have to tell it where references to language-side objects are located in the native code. To overcome this limitation we chose to hack the current VM to better interact with the JIT. More details on this topic follow in the following paragraphs.

NABUJITO Dynamic Code Generation. NATIVEBOOST mainly consists of a visitor over the bytecode-level IR that is provided by the PHARO compiler. Additionally we reimplemented some of the aforementioned helper routines to switch execution context in the VM. The main difficulty of the NABUJITO compiler is the missing interface to the JIT. For instance we did not have direct control on which methods in PHARO are jitted or not, or to force-JIT a method. We added one additional primitive to be able to manually trigger JIT compilation.

For standard methods NABUJITO takes the bytecodes and transforms them with a visitor to native code. It also applies simple optimizations such as creating low-level branches for PHARO-level branching operations such as `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. At this point we benefit from the JIT integration of BENZO itself. As a reminder, we have shown in Section 3.2 how BENZO-enabled methods are treated like normal primitive methods. The VM triggers a BENZO primitive which itself then jumps to the native code attached to the BENZO-enabled method. By default the COG JIT can only directly inline the native code for a known set of primitives. As we have shown in Section 3.2.1 that the COG's JIT was made aware of the special behavior of the BENZO primitive. Hence, whenever a BENZO-enabled method is jitted its native code is directly accessible to the JIT and inlined. Thus we essentially remove the overhead of activating BENZO-enabled methods since we do not have to leave the JIT execution mode. As a result we call BENZO-enabled methods at the same speed as the existing JIT.

Talking to the JIT. After the initial promising progress on building NABUJITO on top of BENZO we soon realized that it does not suffice to just generate the equivalent native code as the VM internal JIT. The first goal was to compile a simple method that just returns a constant integer. Even at this stage it became apparent that there is a missing interface to the JIT. To explain that we have a look at the standard stack frame setup of a jitted method in COG shown in Listing 5.3.

```

1 // push the current framepointer on the stack
2 push EBP
  // use the stack pointer in ESP as new framepointer
4 mov  EBP, ESP
  // push the current jitted method on the stack
6 push 0x1f452b00<CogMethod>
  // move the Pharo object nil to the EBX register
8 mov  EBX, 0x1f500004<nil>
  // push nil (stored in EBX) twice on the stack
10 push EBX
   push EBX

```

Code Example 5.3: COG JIT Stackframe Setup

After finishing executing these setup instructions the stack frame looks as depicted in Figure 5.8. As we can see there are already two references to `nil` in the stack frame header. Already these two references pose a problem in

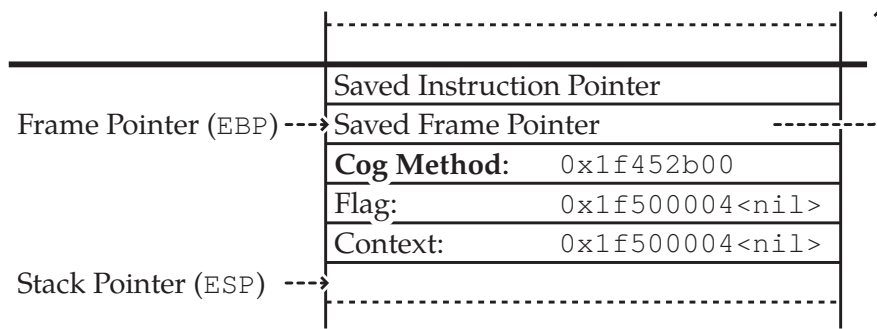


Figure 5.8: Cog Stack Frame Header

a simple `NABUJITO` setup, but for now we focus on the reference to the `CogMethod`. As we explained earlier the `CogMethod` is a meta object at `JIT`-level to make certain information of `PHARO` methods faster accessible. The `VM` currently keeps a pointer to the class, the selector or the number of arguments cached in there. Having the information there improves locality and make the assembler code required for the frequent querying simpler. Going back to the native code in Listing 5.3 we see that we need the final address of the `CogMethod` for the frame setup. However, at the moment where `NABUJITO` generates the native code the target method is not yet jitted. This again implies that the corresponding `CogMethod` has not yet been allocated by the `JIT`. And since the native code has to be installed in the `JIT` we inevitably have to wait for `NABUJITO` to finish compilation, we are stuck.

Instead of directly putting the absolute address of the meta-object in the native code we add a call to a helper routine which will patch the original code on the first activation. We can do so since we know the following things:

- `CogMethod` has a fixed size known upfront,
- we know the relative offset of the jitted method's instruction to the start of its `CogMethod`,
- we can access the instruction pointer with a helper routine.

With this information we modify `NABUJITO` to generate the modified frame setup show in Listing 5.4.

```

1 // push the current framepointer on the stack
  push EBP
3 // use the stack pointer in ESP as new framepointer
  mov  EBP, ESP
5 // move the address of a helper function into EAX
  mov  EAX, 0x643d02e<pushCogMethodHelper>
7 // call the helper method
  call EAX

```

Code Example 5.4: `NABUJITO` Stack Frame Setup

In the `pushCogMethodHelper` we access the instruction pointer from where the call happened in the stack frame setup and deduce the start of the `CogMethod`. Once the address of the `CogMethod` is retrieved the `pushCogMethodHelper` patches the `MOV` and `CALL` instruction in the jitted method. The result is shown in Listing 5.5.

```

1 // push the current framepointer on the stack
2 push EBP
  // use the stack pointer in ESP as new framepointer
4 mov  EBP, ESP
  // patched instructions done by pushCogMethodHelper
6 push 0x1f452b00<CogMethod>
  // patched 'call EAX'
8 nop

```

Code Example 5.5: NABUJITO Patched Stack Frame Setup

By using this indirection we circumvent the missing interface to the JIT. The helper routine only imposes a one-time overhead, however we slow down the final execution of the NABUJITO method by a single `NOF` instruction. Yet, looking at the `COG` stack frame in Figure 5.8 we only dealt with finding the reference to the `CogMethod`. So far we left out the `GC` interaction at JIT-level, which leads us to the following paragraph.

Overcoming the Missing VM Interface for the JIT. `COG` embeds references to normal `PHARO` objects in its jitted methods. Most often this is the case for the symbols used as selectors in message sends. This is different from the indirect approach used in compiled methods for the bytecode interpreter. There all objects used in the method are stored in a separate literal array and referenced by an index. Hence the bytecode can stay very compact and more importantly does not have to be updated on each `GC` pass. While for space consumption was the only concern on the early `SMALLTALK` implementation the JIT only focuses on performance and thus avoids as many indirections possible; hence the use of direct references in the JIT. That implies that unlike the bytecodes, the JIT code is no longer independent of the location of the referenced objects and thus has to be updated on each `GC` pass. Additional to moved objects on the `PHARO` heap, the JIT space itself moves a `CogMethod` when compacting native code. Hence the `GC` has to also be aware of jumps and reference to another `CogMethod` inside the native code. In `COG` this additional information is stored in the `CogMethod` itself, called `method map`. It contains simple entries which describe the location of jumps, calls, references to other `CogMethod` objects and references to `PHARO` objects.

The low-level design of `COG` has significant implications on how NABUJITO has to interact with the VM. NABUJITO has to provide the location of every reference and jump inside the native code. With the design of NABUJITO so far, this

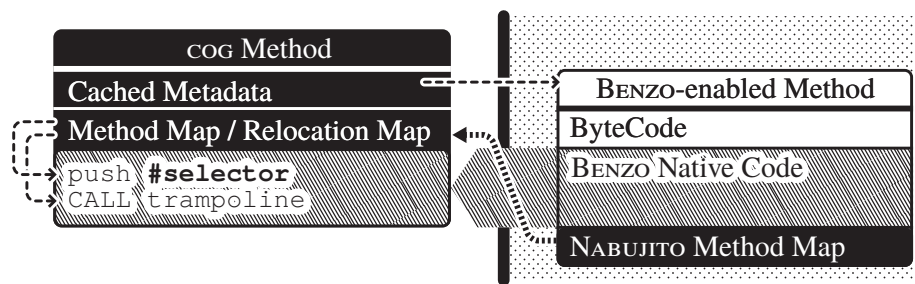


Figure 5.9: NABUJITO can easily transfer a static version of the native code of a PHARO method to the JIT. However, there is no interface to pass the crucial relocation information.

is not directly possible. So far NABUJITO directly copies the native code from the BENZO-enabled method to the `CogMethod`. Hence, NABUJITO ignores all the additional information required for the JIT to work properly. To comply with the JIT we implemented a custom primitive for NABUJITO with custom JIT support, essentially creating a fork of the VM. The newly added primitive is a copy of the existing BENZO primitive with JIT support. However, the NABUJITO adds support for the `CogMethod` relocation maps. At language-side the NABUJITO compiler stores a relocation map as the first literal in the compiled method. In the customized JIT code for the NABUJITO primitive we read this relocation information and forward it to the COG JIT infrastructure. Essentially we replicate the information of the JIT-level `CogMethod` inside a BENZO-enabled PHARO method.

5.2.3 NABUJITO Validation

After explaining the implementation details and challenges of NABUJITO we present a performance validation of our language-side JIT compiler prototype. Our current prototype implementation is not complete yet, we envision that the final compiler will produce the same native code as the existing JIT of the COG VM. Based on that idea we focus our evaluation mainly on the language-side code generation. Even though the underlying BENZO infrastructure caches the native code, the compilation step itself is several magnitudes slower than the native JIT version. Hence we first evaluate in detail the compilation speed of NABUJITO, and only in the second part we focus on the real execution speed of the generated native method.

Compilation Time

In this first part of the performance evaluation for our BENZO-based JIT compiler we focus on the language-side code-generation part. NABUJITO essentially generates the same native code as the VM-level JIT, hence there is

no performance difference at evaluation time. However, NABUJITO is clearly slower during the warm-up phase. Compilation of the native instructions will take considerably more time compared to the VM-level implementation of the same bytecode to assembler transformation. The cost of transforming the bytecodes to native code at VM-level can be measured in native instructions, whereas the unit at language-side is bytecodes. However, we point out again, that this is a one-time overhead. From the in-production experience of NATIVEBOOST, the BENZO-based FFI (see Section 4.3), we know that these costs amortized, especially for long-term applications. Instead of focusing on the final performance of the generated code, we present the compilation time compared to the normal PHARO bytecode compiler, which also resides at language-side.

	Compilation Time [ms]
PHARO Compiler	71 ± 2
NABUJITO	73 ± 2

Table 5.3: Compilation efforts of the standard SMALLTALK compiler in PHARO and NABUJITO for the a simple method returning the constant `nil`.

In Table 5.3 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 5.3 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 4.3 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.

5.2.4 NABUJITO Limitations and Future Work

Hidden VM Internals. The major obstacle found while implementing NABUJITO is the lack of a language-side interface to the JIT. In Section 5.2.2 we al-

ready showed how we circumvented most of the limitations. Our final conclusion was to extend the VM and add a customized primitive with its own JIT support. Strictly speaking this is against the principles of the BENZO framework, where tools should be implemented transparently at language-side. Even though the NABUJITO is mainly implemented in PHARO, the required VM modifications result in the problems already described in Section 3.5.3. VM extensions tend to be less maintainable, eventually NABUJITO will take the same path as many other research VM projects based on PHARO and stay unmaintained until the VM becomes incompatible.

Most of the problems described for NABUJITO are being addressed with SISTA, a new adaptive JIT compiler for the COG infrastructure. Until now the JIT compiler for COG is completely written in SLANG and thus frozen at VM compilation-time. SISTA takes a different approach by implementing most JIT optimizations at language-side. Though the underlying approach is very different from NABUJITO. SISTA will require a new VM that supports querying the status of the inline caches from PHARO code. Based on the retrieved information SISTA will apply standard optimization techniques like inlining. Instead of directly generating native code at language-side SISTA will encode additional information in an extended bytecode set. The VM is then capable of extracting the necessary information to generate optimized native code. SISTA essentially avoids the problems we described in Section 5.2.2 which occur when directly injecting native code into the JIT machinery. SISTA's flexibility lies between the current JIT present in COG and the NABUJITO prototype.

Debugging Cycle. While working on NABUJITO we encountered the same debugging limitations found in the other BENZO applications. However, the interaction with the existing JIT required already substantial debugging efforts, mostly at assembler-level. Hence, BENZO's missing high-level debugging facility does not have a big impact on the general development of NABUJITO. The main issue is that the VM itself lacks separate tests for the JIT infrastructure. Even so the COG branch supports a high-level simulator for running the JIT this is currently not supported under PHARO. Additionally the VM lacks dedicated tests for the separate parts of the JIT infrastructure. However, with the previously mentioned SISTA project, efforts are being made to enable the existing VM debugging infrastructure on PHARO, along with dedicated tests.

Missing Optimizations. One major performance optimization missing in both, the original PHARO 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 or SELF [15] when using a per method JIT. A better match for this

seems to a trace-based JIT such as the one present in PYPY or LUAJIT. 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. The JIT lives on the same level as the information it needs relying on the already present reflective features of SMALLTALK.

5.3 BENZO Applications: Outlook and Summary

In this chapter we presented two BENZO-based research projects: dynamic primitives and a language-side JIT compiler prototype. We have shown in Section 5.1 how the WATERFALL toolchain allows us to modify PHARO primitives on the fly. WATERFALL uses the existing VM source code written in the SMALL-TALK subset SLANG, thus simplifying the modification of existing primitives. Using the same SLANG sources WATERFALL is also capable of compiling whole plugins on the fly. In Section 5.1.4 we showed that WATERFALL outperforms a pure PHARO-based solution when instrumenting primitives. WATERFALL is up to one magnitude slower than native VM primitives leaving room for optimizations.

In the second part we presented NABUJITO a language-side JIT compiler that uses BENZO for the code generation part. Even though NABUJITO looks promising, the current implementation does not go beyond the stage of a prototype. We identified that a reasonable BENZO-based JIT implementation requires a well-defined interface to the otherwise isolated JIT. We have shown that the compilation time from bytecode to native code takes the same time as the standard bytecode compiler. However, from a real-world point of the view the current NABUJITO setup is insufficient since it requires a customized VM. A sufficient stable JIT interface is required to efficiently implement NABUJITO.

Both of these applications are a further validation of the BENZO framework and the concept of an open language-runtime without a clear distinction between language-side and VM-side. With the current setup we are capable of hosting important VM parts such as non-essential plugins and primitives to the language-side. However, again we encountered the typical limitations of the BENZO framework: missing high-level debugging and hard-coded assembler assumptions. As before we refer to the suggested BENZO improvement presented in Section 3.6.

While working on NABUJITO we realized that most important primitives two different implementations. Once there is the default implementation written in SLANG using the C stack. Then there is an additional assembler-level implementation for the JIT. This is necessary to avoid frequent context switches for primitives. However, with the current BENZO infrastructure it might be possible to use the same definition of the primitive for both usages and thus reduce code duplication.

The two BENZO research applications in this chapter have shown the limitations of our framework. This will conclude our validation of BENZO itself and we will focus on the future work to extend BENZO's capabilities in the following chapter.

FUTURE WORK

Chapter 6

Contents

6.1	Background	108
6.2	Language-side Improvements	109
6.3	vm-level Improvements	118
6.4	Summary	119

Introduction

In Chapter 2 we gave an overview of different concepts of reflection focusing on the main distinction between language-side and vm-side reflection. While language-side reflection is very well described in research and rather widespread in dynamic languages, the vm-side counterpart is not. Compile-time reflection is the center of many popular vm generation frameworks, but they usually exclude the dynamic reflection aspect of the final binary. Nevertheless, it is always possible to introspect (structural reflection) the vm at a very basic level. Additionally, there are tools like `DTRACE` which provide a simple way to instrument a binary with little prior setup required. Thus a limited form of intercession is possible on the binary executable themselves. However, this still does not make the internal structural information of the vm accessible that were available at compilation time. And the restrictions are even more severe when it comes to vm-level intercession. It is foreseen for languages to dynamically influence the underlying vm.

In the course of this thesis we presented tools that try to enter the field of vm-level reflection – all based on `BENZO`, a common framework to activate native code from language-side.

`WATERFALL`'s dynamic primitives are a first step towards modifying vm behavior from language-side in a rather controlled way. By bringing the metacircular vm sources alive in `PHARO` we connect the former static definition to the running artifact. Modification happen not by injecting basic native instructions but at high-level by modifying and dynamically compiling primitives.

In contrast to `WATERFALL` we developed `NABUJITO`, a JIT compiler prototype, that moves the original vm component to the language-side. While `NABUJITO` is defined as language-side compiler using familiar coding patterns, its interaction with the vm is not clean. Unlike the plugins and primitives defined by `WATERFALL` the JIT generates native code that is heavily depending on the

low-level and internal execution model of the VM. Unlike WATERFALL NABUJITO requires a modified VM to add a basic interface for manually injecting JIT code.

In this chapter we present possible solutions and an early prototype VM that addresses the limitations we encountered while developing NATIVEBOOST, the BENZO-based FFI, WATERFALL and NABUJITO. We start by listing possible improvements for the language-side part of the BENZO infrastructure such as providing a well-defined high-level intermediate format. This will lead to a description of required VM-level improvements to make applications such as WATERFALL or NABUJITO feasible outside a research context.

6.1 Background

The improvements to the existing infrastructure BENZO and possible future work is influenced by two research projects we described already in detail in Section 2.2.3: the PINOCCHIO VM and the KLEIN VM. For this chapter we present a small summary of these two metacircular VMS with the focus on two things: their own limitations compare to BENZO and their influence on improvements and future work.

6.1.1 PINOCCHIO VM

The PINOCCHIO VM [52] presented in Section 2.2.3 is a direct predecessor of the work presented in this thesis. The knowledge gained while participating on PINOCCHIO had a great influence on the development direction of BENZO and its applications.

Unlike PHARO running on the COG VM the PINOCCHIO research VM has no bytecode interpreter. The only execution base is native code which is directly generated by the language-side compiler. At the current stage of development PINOCCHIO has not yet support for a separate image as in PHARO. The runtime image is currently defined by the bootstrap process where classes, objects and methods are exported into binary images and linked together with a primitive kernel to a final executable.

Going Native. We took from PINOCCHIO that language-side native code generation is not more complex than generating bytetimes. Instead we directly embrace the native world. This means that in the core PINOCCHIO already uses many concepts that are only introduced by the JIT in the COG VM. Hence, PINOCCHIO does no longer distinct between JIT mode and interpreter mode. Here the gain for PINOCCHIO are twofold: we could boost the performance of the language-runtime and simplify the design by not needing a dual compilation pipeline for the JIT and the bytecode.

Going Meta. Even so PINOCCHIO directly uses native code as core execution mode we avoided to directly write native code if possible. For instance the method lookup in COG is statically implemented at VM-side using SLANG. We described in Section 2.2.3 in detail how PINOCCHIO uses language-side code instead for the lookup. Using the combination of low-level code to flatten out meta recursion we still have full language-side control over the lookup while maintaining good performance.

Missing Low-level Reification. The most obvious shortcoming of PINOCCHIO was the lack of its own garbage collector. Instead of investing time into a separate well-defined GC PINOCCHIO relies on the conservative Boehm GC¹ built for C programs. The Boehm GC is sufficiently fast to run PINOCCHIO as a prototype, however, due to its generic nature it is not as efficient as a specific GC. However, PINOCCHIO lacked the necessary reification at level of the object layout to properly implement a GC. All the notion about the object layout in memory are hard-coded in the compiler in several places.

Missing C Independence. The second negative point of PINOCCHIO is its dependence from C. During the course of the PINOCCHIO development we greatly reduced the quantity of C code. However, for simplicity we relied on a small C Kernel for the complete bootstrap of the language. Additionally some crucial primitives that required system calls were implemented in C.

6.2 Language-side Improvements

In this section we present the suggestion for improvements related mostly tied to the language-side part of BENZO. Most of the solutions have been presented in the separate chapters of BENZO in Section 3.6, NATIVEBOOST FFI in Section 4.5 and the BENZO application prototypes (Section 5.1.5 and Section 5.2.4).

6.2.1 Improved Domain Specific Inspectors

Domain specific inspectors are important for an efficient development. Similar to the JIT approach we have to optimize the frequent tasks during development and provide a seamless integration. This becomes even more important when working with low-level code and data that does not come with existing first-class structures. We noticed that using BENZO for NABUJITO and WATERFALL that it is more convenient to rely on an existing low-level text-based debugger such as gdb to inspect C-level structures.

¹http://www.hpl.hp.com/personal/Hans_Boehm/gc/

We have seen excellent use of inspectors in the VM development of the COG VM itself. The original simulator supports inspecting objects in simulated raw memory. COG added additional inspectors including disassembled instructions for the JIT development. However, with the recent changes the advanced simulator does not yet run in PHARO and requires attention.

Lately we have seen a very similar approach for the MAXINE research VM [55]. It provides excellent low-level debugging interaction, switching seamlessly between low-level assembler views and high-level object inspectors. We believe that it is an imperative requirement for a VM development IDE to support customizable inspectors that span from high-level to low-level. Even though existing C-focused IDEs provide more and more support for integrated inspectors the VM domain has different needs. C inspectors are tailored towards fixed-sized objects whose types can be statically inferred. Whereas, for VMS we only have a handful types of object layouts and the real type is only implicitly available. For instance in certain VMS the class is encoded in the header of an object instead of a simple full pointer to the class object. This means that a minor interpretation pass is necessary to retrieve such information. Which is why most C-focused IDEs are only partially sufficient for an efficient VM development.

6.2.2 Virtual CPU an Assembler DSL

BENZO uses ASMJIT as assembler backend which is currently limited to x86 instruction set. This choice is aligned with PHARO's main development focus for the most common operating systems. However, this choice of architecture already excludes mobile devices which currently focus on ARM-based architectures. To support this new architecture we have to extend the assembler backend in ASMJIT. While the implementation effort for a new ASMJIT architecture is an implicit requirement the implications on BENZO-based applications are more severe. For instance we already mentioned in Section 4.5.2 how the NATIVEBOOST FFI would suffer from code duplication. Essentially each newly added CPU architecture has ripple-effects up to the final BENZO-based applications. Each BENZO application has to duplicate all native usage in and create internally separate generates for each platform. We believe that this approach is not the right path as it forces BENZO users into code duplication.

A much better solution is to provide a more abstract and platform independent low-level intermediate format at BENZO-level. Ideally we can push the platform specific code generation fully to the native backends in BENZO itself. BENZO-based applications only have to focus on a single low-level instruction format reducing the development effort.

We also noticed that for actual testing the native instruction set is not op-

timal. For instance the x86 instruction with all its modifier codes and variable width instructions makes it tedious to implement a proper simulator. Writing bindings for an existing simulator such as Bochs² is a better choice. However, even with a proper simulator ready for x86 we can not provide a very fluent debugging process. For instance, the way assembler instructions are written in BENZO require them to be fully generated before they can be interpreted. We are used from PHARO that any code snipped is executable, a property that we would like to bring to the low-level development as well.

VIRTUALCPU: A Low-level Intermediate Format. To reduce the aforementioned platform dependency of BENZO we developed a intermediate low-level representation called VIRTUALCPU. It is based on a three-address-code (TAC) to simplify the adoption of optimizations such as static single assignment (SSA) [21]. Additionally we chose to postpone register allocation to the final code generation phase. By using a TAC-based format and rewiring the internals we are even able to make the VIRTUALCPU code directly executable in PHARO.

Before we go into the implementation details of VIRTUALCPU we show it is used. VIRTUALCPU is based on TAC instruction which take the following form:

```
result := argument1 OP argument2
```

There are three operands involved, `result`, `argument1` and `argument2`, from which the name of this TAC format originates. Based on this assumption, each standard VIRTUALCPU instruction returns a temporary variable which can be used for further operations. This makes the information-flow much more consistent. For instance the x86 instructions which sometimes have a predefined result register and sometimes not.

The following code example outlines the basic usage of VIRTUALCPU:

```
Benzo vcpu x86 generate: [ :cpu | | temp1 temp2 |
    temp1 := cpu memoryAt: 16r12345.
    temp2 := cpu uint: 2.
    cpu return: temp1 + temp1 ]
```

Code Example 6.1: Basic VIRTUALCPU Example

Which corresponds to the same functionality expressed in the following x86 instructions:

```
Benzo x86 generate: [ :asm |
    asm mov: 16r12345 ptr to: asm EAX.
    asm add: asm EAX with: 2.
    asm return ]
```

²<http://bochs.sourceforge.net/>

For this basic example we see that the two formats do not differ very much. Though, already on the example of the `return` instruction it becomes obvious that the TAC-based solution is more explicit. When using more complex control structures the difference is apparent:

```
Benzo vcpu x86 generate: [ :cpu || a b c |
  a := cpu uint: 1.
  b := cpu uint: 2.
  c := cpu uint: 3.
  (a = b and: b = c)
    ifTrue: [ c value: 5 ]
    ifFalse: [ c value: 10 ] ]
```

VIRTUALCPU benefits from using explicit instruction objects to add a PHARO-like DSL on top. The previous example looks fairly similar in plain PHARO code:

```
| a b c |
a := 1.
b := 2.
c := 3.
(a = b and: b = c)
  ifTrue: [ c := 5 ]
  ifFalse: [ c := 10 ].
```

The DSL is transparently implemented by adding PHARO methods on the corresponding VIRTUALCPU instructions. Under the hood VIRTUALCPU will lower the TAC instructions to low-level ASM instructions for the ASMJIT backend.

VIRTUALCPU Implementation Overview. So far we presented the external interface of the VIRTUALCPU format which works similar to the existing BENZO assembler format. We will now shed lights on the internal implementation details of VIRTUALCPU which is divided in three classes of objects: CPUS, low-level objects and instructions. The relationship between these main classes are shown in Figure 6.1.

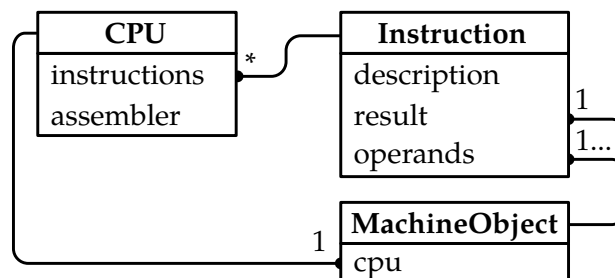


Figure 6.1: VIRTUALCPU Overview

CPUS: Source for values named `cpu` in the previous examples. The `cpu` object will contain the list of instructions and for which backend it should gen-

erate the native code. Additionally we provide specialized CPU objects for debugging purposes or even immediate evaluation.

Low-level Objects or Values: A category of values or helper objects. For example `cpu uint: 1` will create low-level word that contains the value 1. The corresponding high-level equivalent would be a variable. In the final native code such a value object might be mapped to a register or stack location. Another example is the special low-level object that is the result of a comparison, for instance created by `a = b`. This object holds the result of the comparison and implements the known boolean messages such as `ifTrue:ifFalse: or and:.`

Instructions: Encapsulates an instruction type, operands and result value according to the TAC format. Much like in PHARO the programmer will almost never directly interact with instructions but with the yielded value. For example `temp1 + temp1` yields a new VIRTUALCPU value and internally records an add instruction in the CPU object that created the value in `temp1`.

In VIRTUALCPU the main responsibility for the programming interface lies on the machine objects. In contrast to that we see that in ASMJIT the full interface is defined on the ASM itself. The intermediate values are almost never used, and even registers only play the role of spectators. Hence, VIRTUALCPU takes full advantage of the different types of values to define a simple DSL. This way we are able to for instance create branches and loops following PHARO semantics rather low-level jumps and labels.

cpu Types. Typically, the VIRTUALCPU programmer will only directly interact with the machine objects. VIRTUALCPU instructions are tracked in the related CPU object, but usually not accessed directly. However, the CPU objects play an important role in the development process. By default each operation on machine objects is dispatched over the corresponding CPU object as shown in the following code example for the + operation.

```
MachineObject >> + machineObject
  ^ cpu add: self with: machineObject
```

So, in the example `c := a + b`, the result object `c` is created in the CPU object by invoking the `add:with:` method. This allows us to easily customize the behavior of the CPU objects. Currently, we have two types of CPU objects in use:

Generating cru: This CPU delegates the addition to a low-level builder which keeps track of the corresponding TAC operation. After completing a routine, it will compile the TAC instruction to native code using the specific ASM backend.

Evaluating CPU: This CPU object does not keep track of the TAC operations but directly evaluates them. Typically we use this CPU type for debugging purposes. The current setup includes a byte array simulating the native memory. The evaluating CPU gives us a PHARO-like debugging behavior with very little additional implementation costs.

VIRTUALCPU Optimizations. To get to the final native instructions the VIRTUALCPU infrastructure compiles the high-level TAC instructions to the specific backend. The current compiler is divided into the following passes:

- Platform Specific Transformation
- Register Allocation
- Superfluous Assignment Remover
- Platform Specific Assembler

Currently VIRTUALCPU does not include more aggressive optimization techniques such as constant folding or subexpression elimination that are associated with a TAC IR. The idea is to move the existing BENZO applications to this new VIRTUALCPU format and share the improvements across all tools.

Custom Machine Objects. Using first-class machine objects during for the native code format in VIRTUALCPU has a secondary application that is not immediately visible. At code generation time we can use other machine object than the ones described so far. This finds a direct application in the NATIVEBOOST FFI when working with structs. Instead of manually delegating the code generation for accessing fields of a struct to a mirror object we can use it naturally inside VIRTUALCPU code. The following code example outlines this idea.

```
Benzo vcpu x86 generate: [ :cpu |
  | address struct fieldValue |
  address      := cpu address: 16r1234.
  struct       := MyStructure pointer: address.
  fieldValue := struct field1 ]
```

In this example the struct field1 message will create several instructions hidden from the user. Typically, this involves dereferencing the pointer and masking out the corresponding bits of field1 from the memory location. The only visible artifact from the outside is the returned result.

6.2.3 Barrier-free Low-level Interaction

Shifting from VM development to the final language-runtime we see a similar issue when it comes to tools that span abstraction levels. It is not directly possible to inspect low-level objects from language-side. Focusing the on the BENZO architecture what comes closes to inspecting low-level objects is the

`struct` support for the `NATIVEBOOST_FFI` library described in Section 4.2.4. We already use this approach for the `NABUJITO` project to inspect `VM` internal meta objects for debugging purposes. It is important to note that giving access to the `VM` internal objects is not permitted in most languages. The previously mentioned `VM` generation frameworks usually have first-class objects for all the `VM` internal objects or provide mirror-like facilities to access objects from raw memory. Usually, none of this structural information survives the `VM` compilation phase. Essentially this leaves the final `VM` binary with little or no means for introspection. Of course the same restrictions apply then for language-side tools like `BENZO` that want to interact with the `VM` internals.

Customized `VM` `MOP`. We have seen in Section 5.2.4 for `NABUJITO` that the only way to circumvent such issues is by creating modified `VMs` which enable specific interaction points. There are other `PHARO`-based research projects [6, 33] that took the same path and created a modified `VM`. We, believe that with an extended low-level `MOP` the focus for research projects could shift from the `VM` to the language-side. The final extreme is to have a system that works like the described `KLEIN VM` where there is no longer a clear distinction of what is `VM`-level and what is language-side.

Anticipated Debugging. For `BENZO` we have more modest intermediate goals. The major drawback for a seamless developer experience is the lack of a dedicate low-level debugging infrastructure. At this point, `BENZO` developers have to rely on 3rd-party C-centric tools for debugging. Hence, a developer has to decide upfront at which abstraction level the debugging should occur. Either at high-level without the possibility to deal with low-level errors, or at low-level losing all the inspection capabilities. Besides the shortcomings that either side of the decision will bring, already the fact that the debugging direction has to be anticipated is inappropriate.

We outlined in Section 3.6.2 already several ways to improve the current debugging situation for `BENZO`. The most important focus is on reducing the cases where the programmer has to anticipate the debugging tool. Since we have to deal with two very distinct abstraction levels we can not only rely on a pure language-side solution to provide different debuggers [18]. The major problem is the serious implications of a low-level error. Unlike user-level errors they are not well-defined or even contained. It is astonishingly simple to corrupt the `VM` memory while writing low-level code and thus breaking any contract with the `VM` code. However, it is more common to access protected memory due to a wrongly dereferenced pointer. Hence, the `BENZO` should focus on this most common bug by following the solution outlined in Figure 6.2.

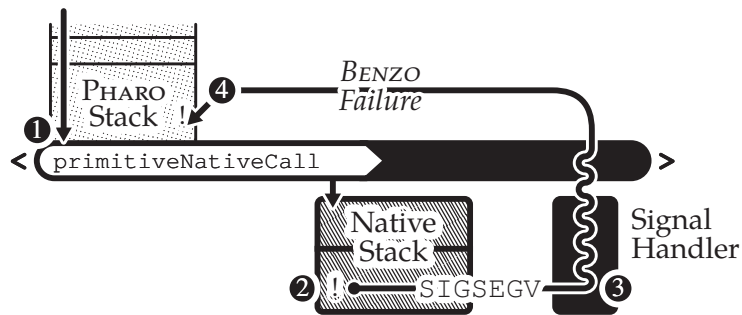


Figure 6.2: BENZO Debugger Outline

1. Standard PHARO method activating a BENZO-enabled method through the `primitiveNativeCall` primitive.
2. Native code causing a memory access violation (for example `SIGSEGV`) which can not be handled by PHARO directly.
3. Low-level signal handler is activated by the operating system and tries to walk back the native stack up to the `primitiveNativeCall` activation.
4. After successfully finding the `primitiveNativeCall` the signal handler sends a BENZO failure back to PHARO.

Missing Barrier-free Debugging. After proposing a solution to improve BENZO's bug recovery behavior we immediately encounter a second problem. How do we debug low-level code? With the aforementioned solution we are able to recover from certain low-level errors and signal them properly at language-side. In a SMALLTALK-like environment the debugger will pop up on the location causing the error and thus allowing a programmer to inspect stack and variables. To provide the same facility for BENZO we have to plug into the existing low-level debugging utilities such as `ptrace` to enable stepping over native instructions. The following Figure 6.3 outlines the basics of a debugger that crosses the high-level / low-level barrier.

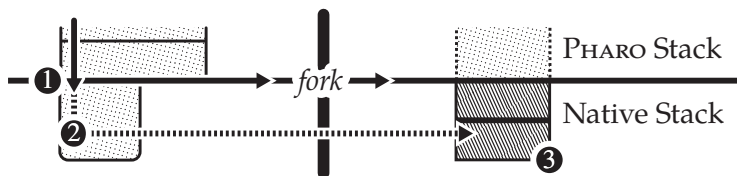


Figure 6.3: BENZO Crossover Debugger Outline

1. Point where a BENZO initiates a native call and the debugger switches from a high-level PHARO stack to the low-level native stack. To properly use low-level debugging tools we fork the complete VM process.

2. The low-level debugger in PHARO switches the underlying debugging interface. Instead of directly interacting with first-class PHARO context we communicate to the the forked process with tools like `ptrace`.
3. The forked process is updated according to the actions initiated from the PHARO side.

The outlined debugger will not work in certain cases where the native code directly interacts with the outer image. The forked debugger process provides security from the main PHARO image by isolating it. However, for many BENZO applications such as the FFI implementation this limitation would not apply.

6.3 vm-level Improvements

We have presented solutions to improve the code quality and the development experience when working with BENZO that are focused on the language-side. We concluded the previous section that the VIRTUALCPU intermediate format brings a certain level of platform independence to native code written with BENZO. At even higher-level, we pointed out how the lack of domain specific inspectors prevents a seamless development experience. The last of the three points present was barrier-free low-level interaction from the language-side point of view. We mainly focused on the debugger interaction when working with low-level BENZO code. One part included the fact that most debugging actions have to be anticipated when working with BENZO. This is counter-intuitive to the default PHARO development workflow. However, the solution we presented already required certain vm-level support to become feasible.

It becomes obvious that a more powerful BENZO implementation also requires modification at vm-level. We explored the limitations of BENZO with NABUJITO JIT compiler presented in Section 5.2. It is not possible to build a JIT compiler purely on top of BENZO, instead we had to fall back on a customized vm with the necessary modifications. The goal of this section is to outline vm-level improvements to push the envelope on dynamic changes that extend down to the vm. We mainly focus on the reification of vm-level structure beyond compilation time. In this sense we follow the findings presented for the KLEIN VM discussed in the related work Section 6.1 of this chapter.

6.3.1 Missing vm-level Reification

With BENZO we chose a non-invasive approach to make the low-level power accessible to the language-side. BENZO is built around the simple capability to dynamically invoke native code. Even though this is the basis for generic interaction with the vm it is too unstructured. We have seen that for many BENZO applications this is not a serious restriction. Yet, in the case where we want to interact with the vm internals without modifying the vm we need better access. We roughly distinct between two types of vm-level access that require proper reification.

Static Part: When building metacircular vms using an intermediate language such as C, the original high-level structure is lost. It is possible to partially reconstruct the internal structures from the low-level debugging information, such as DWARF.

Dynamic Part: Next to the static part of the vm there is a more dynamic aspect linked to the language-side. In the case of C-based vm these are all

the places where native code and memory is used directly.

We addressed some of the access earlier in this thesis. For instance `NATIVEBOOST` features a `VM` proxy objects that exposes a public interface to the `VM`. However, this does not cover real reflective access of the `VM` internal structures. In Section 5.1 we have shown how `WATERFALL` reuses the sources of the metacircular `VM` to generate primitives. This immediately makes structural information accessible at runtime. For a truly self-aware system the situation we found in the `PHARO VM` is not sufficient.

6.4 Summary

We have shown in previous chapters that the dynamic high-level low-level programming framework `BENZO` is a valid option for a set of typical `VM`-level components. However, its minimalist approach to generate native code at language-side has clear limitation as we have shown `NABUJITO` in Section 5.2. Additionally `BENZO`'s current implementation lacks a seamless debugging process. In this chapter we have outlined solutions for these problems. Namely, we see that for an extended set of `BENZO`-based applications we need sufficient reification at `VM`-level. On the language-side part of `BENZO` we see the strong requirement of a more abstract representation of the native code to build better development tools.

CONCLUSION

Contents

7.1 Contributions	122
7.2 Published Papers	122
7.3 Software Artifacts	124
7.4 Impact of the Thesis	125

Introduction

In this chapter we summarize this dissertation. We list the contributions, the published papers and the created software artifacts and their impact.

Chapter 2 listed related work for this dissertation. We presented different types of metacircular high-level language vms. As a result we found only published results of two research vm that have a unified model. Most other vms follow a clear separation between vm-side and and isolated language-side.

Chapter 3 described a high-level low-level programming framework named `BENZO`. The core functionality of `BENZO` is to dynamically execute native-code generated at language-side. `BENZO` allows us to hoist typical vm plugins to the language-side. Furthermore we show how code caching makes `BENZO` efficient and users essentially only pay a one-time overhead for generating the native code.

Chapter 4 presented a `NATIVEBOOST`, a stable foreign function interface (`FFI`) implementation that is entirely written at language-side using `BENZO`. `NATIVEBOOST` is a real-world validation of `BENZO` as it combines both language-side flexibility with vm-level performance. We show in detail how `NATIVEBOOST` outperforms other existing `FFI` solutions on `PHARO`. However, we observe that the underlying `BENZO` requires more effort to simplify low-level debugging and improve platform independence.

Chapter 5 focused on two further `BENZO` applications: dynamic primitives and a language-side jit compiler. The dynamic primitives show how a metacircular infrastructure can be used dynamically and at runtime to reify part of the vm necessary for compiling primitives. With the dynamic primitives we are able to alter a part of the execution semantics embedded into the language. This concept is taken further with the

language-side JIT presented in the second part of this chapter. With the JIT we are able to control further aspects of the execution. We fully rely on language-side generated code instead of bytecodes which have tied interactions with the VM. VM-level JIT, however, it is currently limited to simple expressions. Our JIT shows that for certain applications a well-define interface with the low-level components of the VM is required.

Chapter 6 summarized the limitations of BENZO and its application, furthermore we list undergoing efforts on the BENZO infrastructure and future work. We conclude that the BENZO approach for VM interaction requires more support at VM-level. Namely, missing dynamically accessible VM-level reification make it hard to communicate with VM internal components. Based on this observation we present a summary of another research VM which also tries to overcome the limitations identified with the BENZO framework itself. Namely, the project is built around inspection and outside interaction during the complete development process.

7.1 Contributions

The main contributions of this thesis are:

- Description of the properties of an open and reflective language runtime.
- Implementation of BENZO, a dynamic high-level low-level programming framework for PHARO.
- An in depth validation of BENZO with NATIVEBOOST, foreign function interface implemented on top of BENZO. NATIVEBOOST proves the feasibility and efficiency of a dynamic high-level low-level programming framework.
- A BENZO-based language-side JIT compiler showing the boundaries of BENZO.
- A road map for the future, bottom-up implementation of an open language runtime with full VM-level reification using a platform independent IR for low-level programming.

7.2 Published Papers

Flexible object layouts: enabling lightweight language extensions by intercepting slot access

Slots and Layouts as described in this paper written as collaborator with Toon Verwaest are used in PHARO 3.0 and newer. The original implementation pre-

sented in this paper was implemented in an older PHARO 1.0 image and was ported to PHARO 3.0 in 2013.

Abstract: *Programming idioms, design patterns and application libraries often introduce cumbersome and repetitive boilerplate code to a software system. Language extensions and external DSLs (domain specific languages) are sometimes introduced to reduce the need for boilerplate code, but they also complicate the system by introducing the need for language dialects and inter-language mediation. To address this, we propose to extend the structural reflective model of the language with object layouts, layout scopes and slots. Based on the new reflective language model we can 1) provide behavioral hooks to object layouts that are triggered when the fields of an object are accessed and 2) simplify the implementation of state-related language extensions such as stateful traits. By doing this we show how many idiomatic use cases that normally require boilerplate code can be more effectively supported. We present an implementation in SMALLTALK, and illustrate its usage through a series of extended examples.*

Authors: Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard and Oscar Nierstrasz.

Revenue: In Onward! 2011, Reno/Tahoe, Nevada, USA, 2011.

URL: <http://dx.doi.org/10.1145/2048066.2048138>

Language-side Foreign Function Interfaces with NATIVEBOOST

This paper served as the basis for Chapter 4 and discusses the BENZO-based FFI implementation NATIVEBOOST in detail. NATIVEBOOST is used in production in the PHARO 2.0 and newer.

Abstract: *Foreign-Function-Interfaces (FFIs) are a prerequisite for close system integration of a high-level language. With FFIs the high-level environment interacts with low-level functions allowing for a unique combination of features. This duality has a strong impact on the implementation of the FFI: it has to be flexible and fast at the same time. We propose NATIVEBOOST a language-side approach to FFIs that only requires minimal changes to the VM. NATIVEBOOST directly creates specific native code at language-side and thus combines the flexibility of a language-side library with the performance of a native plugin.*

Authors: Camillo Bruni, Luc Fabresse, Stéphane Ducasse and Igor Stasenko.

Revenue: In International Workshop on SMALLTALK Technologies, Annecy, France, 2013.

URL: <http://hal.inria.fr/hal-00840781>

7.3 Software Artifacts

During the work on this dissertation we produced several software artifacts. Many projects emerged out of improving the infrastructure around the PHARO development required to implement the core artifacts presented in this thesis.

Collaboration on First-class Layouts and Slots: In a collaboration with Ton Verwaest (SCG, Switzerland) we built a first implementation of first-class layouts and slots in a SMALLTALK system [52]. In Collaboration with Martin Días (RMOD, INRIA) this initial version was ported to PHARO and is now used in the current release candidate PHARO 3.0¹.

ASMJIT 64-bit Assembler: To reuse the original research compilation pipeline built with PINOCCHIO [13,51] a 64-bit extension was necessary to the initial ASMJIT implementation for PHARO². The extension is included in the current stable PHARO release 2.0³.

Collaboration on the WATERFALL Dynamic Primitives: We collaborated on Guido Chari's (UBA, Argentina) WATERFALL Dynamic Primitive compiler, which resulted in paper currently under submission [16]. The implementation is a prototype and is not used in production.

Collaboration on the Mate VM Prototype: In collaboration with Guido Chari (UBA, Argentina), Javier Pímas (UBA, Argentina) and Clement Bera (RMOD, INRIA) several stages of a prototype VM were built. The implementation mainly follows the concept of an dynamic language runtime which controls every aspect at language-side. The current language runtime is in a early prototype phase that allows us to explore new VM and language concepts, however it is not production ready. Guido Chari will further explore new concepts of Mate in his Ph.D. Clement Bera, after finishing his engineering contract at RMOD, will continue to work as a Ph.D. on the same system.

VIRTUALCPU Compilation Toolchain: In collaboration with Clement Bera (RMOD, INRIA) and Igor Stasenko (RMOD, INRIA) we built a prototype compilation toolchain based on the original work of Pinocchio. The current implementation is a working prototype. Plans exist to integrate a streamlined version into PHARO to server as a platform independent backend to our BENZO-based FFI implementation used in PHARO.

NABUJITO Language-side JIT Compiler: As a third case study for the BENZO framework we implemented a language-side JIT compiler. The current

¹<http://files.pharo.org/image/30>

²<http://smalltalkhub.com/#!/~Pharo/AsmJit>

³<http://files.pharo.org/image/20>

implementation is a prototype that is capable of directly transforming simple methods to executable code. Unlike its VM-level counterpart it is based as a simple visitor over the intermediate bytecode format already present at language-side.

Inspector Framework for PHARO: An important part of reifying concepts is the possibility to inspect and manipulate these objects. We wrote together with Clement Bera a new inspector framework which is used in the latest PHARO release. It allows to quickly define new views on domain objects, an indispensable requirement for interacting with complex data objects. Next to the everyday usage in PHARO it is actively used for the Mate VM prototype where we need transparent access to internal structures of the VM.

Command Line Test Interface for PHARO: In order to perform continuous integration in a maintainable fashion we developed a new modular command line interface for PHARO. It is used in production on the PHARO build server⁴ alongside with simple installer scripts⁵.

Validation Framework: In order to improve the integration life cycle of PHARO we developed together with Benjamin van Ryseghem and Erwann Douaille a validation framework. Many changes in the core PHARO were required to support the previously presented tools. To encourage faster integration we validate each proposed change by running lint rules and all unit tests. The tool generates a validation report as separate website and interacts with the issue tracker.

7.4 Impact of the Thesis

Many engineering artifacts built during this dissertation are used in production for PHARO. The inspector framework allowed us to create many specific views on common objects in PHARO, improving the development in PHARO. For instance the inspectors are tightly integrated into the debugger. Outside the image, we have contributed to improve the PHARO continuous integration work-flow. Identifying the limitations of the BENZO framework will trigger more specific development efforts to improve the debugging capabilities.

Our dissertation helped to bootstrap the MATE research VM which eventually will cope with the limitations of the BENZO approach.

⁴<http://ci.inria.fr/pharo/>

⁵<http://get.pharo.org/>

Bibliography

- [1] Openjdk community. graal project, 2012. 3
- [2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. 18
- [3] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan F. Hummel, Janice C. Sheperd, and Mark 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. 30, 47
- [4] Anders Andersen. A note on reflection in python 1.5. In *Lancaster University*, 1998. 46
- [5] Andrew W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998. 87
- [6] Jean-Baptiste Arnaud, Stéphane Ducasse, and Marcus Denker. Handles: Behavior-Propagating First Class References For Dynamically-Typed Languages. *Science of Computer Programming*, November 2013. 10, 48, 115
- [7] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. 94
- [8] Mark B. Ballard, David Maier, and Allen W. Brock. QUICKTALK: a Smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986. 49
- [9] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009. 42, 47, 94
- [10] Stephen M. Blackburn, Perry Cheng, and Kathryn 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. 18, 48
- [11] Carl F. Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy’s tracing JIT compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of*

- Object-Oriented Languages and Programming Systems*, pages 18–25, New York, NY, USA, 2009. ACM. 20
- [12] Carl F. Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the future in one week – implementing a Smalltalk vm in PyPy. *Self-Sustaining Systems*, pages 123–139, 2008. 19, 47
- [13] Camillo Bruni. Optimizing Pinocchio. Master’s thesis, University of Bern, January 2011. 124
- [14] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Luc Fabresse. Language-side Foreign Function Interfaces with NativeBoost. In *International Workshop on Smalltalk Technologies*, Annecy, France, September 2013. 39
- [15] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, October 1989. 104
- [16] Guido Chari, Diego Garbervetsky, Camillo Bruni, Marcus Denker, and Stéphane Ducasse. Waterfall: Primitives Generation on the Fly. Technical report, INRIA, Sep 2013. Preprint: <http://hal.inria.fr/hal-00871353>. 82, 89, 124
- [17] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The Meta-Helix. In *Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, ISOTAS ’96, pages 157–172, London, UK, UK, 1996. Springer-Verlag. 84
- [18] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. Towards a moldable debugger. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA ’13, New York, NY, USA, 2013. ACM. 115
- [19] David Chisnall. Smalltalk in a C world. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST ’12, New York, NY, USA, 2012. ACM. 24
- [20] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE ’05, pages 46–56, New York, NY, USA, 2005. ACM. 14
- [21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. 111

- [22] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Comput. Lang. Syst. Struct.*, 32(2-3):125–139, 2006. 8
- [23] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings POPL '84*, Salt Lake City, Utah, January 1984. 14, 95
- [24] Kathleen Fisher, Riccardo Pucella, and John Reppy. Data-Level interoperability. In *Electronic Notes in Theoretical Computer Science*, 2000. 49
- [25] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, Eliot, and Sergey 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. 1, 3, 18, 24, 25, 29, 30, 48
- [26] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: a substrate for managed runtime environments. *SIGPLAN Not.*, 45:51–62, March 2010. 14
- [27] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. 83
- [28] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *OOPSLA'97: Proceedings of the 12th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–326. ACM Press, November 1997. 16, 17, 32, 47, 83
- [29] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The JVM is not observable enough (and what to do about it). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages, VMIL '12*, pages 33–38, New York, NY, USA, 2012. ACM. 11
- [30] Stephen Kell and Conrad 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. 11, 20, 24, 49
- [31] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 46
- [32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01*, pages 327–353, London, UK, UK, 2001. Springer-Verlag. 10

- [33] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award. 48, 115
- [34] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987. 8, 30, 46
- [35] J. Malenfant, M. Jacques, and F. N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the First International Conference on Reflection, Reflection'96*, 1996. 9, 46
- [36] Harlan McGhan and Mike O'Connor. PicoJava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, October 1998. 14
- [37] Eliot Miranda. Context management in VisualWorks 5i, 1999. 42, 94
- [38] Eliot 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. 32, 42, 94
- [39] Philip A. Nelson. A comparison of pascal intermediate languages. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, SIGPLAN '79*, pages 208–213, New York, NY, USA, 1979. ACM. 12
- [40] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *Proceedings of the 5th international conference on Generative programming and component engineering, GPCE '06*, pages 49–58, New York, NY, USA, 2006. ACM. 49
- [41] Armin Rigo and Samuele 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. 19, 47
- [42] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 47–65. Springer, 2007. 10
- [43] G. Stefan, A. Paun, V. Bistriceanu, and A. Birnbaum. DIALISP - a LISP machine. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84*, pages 123–128, New York, NY, USA, 1984. ACM. 14
- [44] Éric Tanter. Contextual values. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS '08*, pages 1–10, New York, NY, USA, 2008. ACM. 10

- [45] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, volume 38 of *OOPSLA '03*, pages 27–46, New York, NY, USA, November 2003. ACM. 10
- [46] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 188–197, New York, NY, USA, 1984. ACM. 14
- [47] David Ungar and Randall B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, New York, NY, USA, 2007. ACM. 47
- [48] David Ungar, Adam Spitz, and Alex 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. 13, 23, 47, 50
- [49] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45:59–72, October 2010. 46
- [50] Toon Verwaest. *Bridging the Gap between Machine and Language using First-Class Building Blocks*. Phd thesis, University of Bern, March 2012. 13, 25, 79
- [51] Toon Verwaest, Camillo Bruni, David Gurtner, Adrian Lienhard, and Oscar Nierstrasz. Pinocchio: Bringing reflection to life with first-class interpreters. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 774–789, New York, NY, USA, 2010. ACM. 10, 124
- [52] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. *SIGPLAN Not.*, 46(10):959–972, October 2011. 21, 108, 124
- [53] Jan Vraný, Jan Kurš, and Claus 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. 25, 46
- [54] Christian Wimmer and Michael Franz. Linear scan register allocation on SSA form. In *Proceedings of the 8th Annual IEEE/ACM International*

Symposium on Code Generation and Optimization, CGO '10, pages 170–179, New York, NY, USA, 2010. ACM.

- [55] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. 18, 30, 47, 110

APPENDIX

Chapter

A

A.1 PHARO Programming Language

PHARO is a SMALLTALK inspired object-oriented and dynamically-typed general-purpose language with its own programming environment. The language has a simple and expressive syntax which can be learned in a few minutes. Concepts in PHARO are very consistent, everything is an object: classes, methods, numbers, strings, even the execution context.

PHARO runs on top of a bytecode-based *virtual machine*. Development takes place in an *image* in which all objects reside. All these objects can be modified by the programmer, this includes classes and methods. Hence, we eliminate the typical edit/compile/run cycle and instead incrementally add, remove or modify classes and methods. It is worth noting that *all* classes can be extended with new methods in PHARO. For instance, one can add new operations on integers or strings, classes that are treated as unchangeable internal objects by many other high-level languages. For deployment and debugging, the state of a running image can be saved at any point and subsequently restored.

A.1.1 Minimal Syntax

Reserved Words

<code>nil</code>	the undefined object
<code>true, false</code>	boolean objects
<code>self</code>	the receiver of the current message
<code>super</code>	the receiver, in the superclass context
<code>thisContext</code>	the current invocation on the call stack

Literal Object Syntax

<code>'a string'</code>	
<code>#symbol</code>	unique string
<code>\$a</code>	the character <code>a</code>
<code>12 2r1100 16rC</code>	integers twelve in decimal, binary and hexadecimal encoding
<code>3.14 1.2e3</code>	floating-point numbers
<code>#(abc 123)</code>	literal array containing the symbol <code>#abc</code> and the number <code>123</code>
<code>#[12 16rFF]</code>	literal byte array containing the bytes/integers <code>12</code> and <code>255</code>
<code>{foo . 3 + 2}</code>	dynamic array built from 2 expressions

Reserved Characters in Expressions

<code>"a comment"</code>	
<code>.</code>	expression separator (period)
<code>;</code>	message cascade (semicolon)
<code>:=</code>	assignment
<code>^</code>	return a result from a method (caret)
<code>[:p <i>expr</i>]</code>	code block with a parameter
<code> foo bar </code>	declaration of two temporary variables
<code><pragma>, <primitive: 3></code>	pragma or annotations used in methods, for instances to declare a primitive method.

A.1.2 Message Sending

A method is called by sending a message to an object called the *receiver*. Each message returns an object. Messages are modeled from natural languages with a subject a verb and complements. There are three types of messages with descending precedence: unary, binary, and keyword.

Unary messages have no arguments.

```
Array new.
```

The first example creates and returns a new instance of the `Array` class, by sending the message `new` to the class `Array` that is an object.

```
#(1 2 3) size.
```

The second message returns the size of the literal array which is `3`.

Binary messages take only one argument and are named by one or more symbol characters.

```
3 + 4.
```

The `+` message is sent to the integer object `3` with `4` as the argument.

```
'Hello', ' World'.
```

In the second case, the string `'Hello'` receives the message `,` (comma) with the string `' World'` as the argument.

Keyword messages can take one or more arguments that are inserted in the message name.

```
'Smalltalk' allButFirst: 5.
```

The first example sends the message `allButFirst:` to a string, with the argument `5`. This returns the string `'talk'`.

```
3 to: 10 by: 2.
```

The second example sends `to:by:` to `3`, with arguments `10` and `2`; this returns a collection containing `3, 5, 7, and 9`.

A.1.3 Precedence

There is a fixed global precedence when evaluating expressions in PHARO: Parentheses > unary > binary > keyword, and finally from left to right.

```
(10 between: 1 and: 2 + 4 * 3) not
```

Here, the messages `+` and `*` are sent first, then `between:` and `:` is sent, and finally `not`. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*, so `2 + 4 * 3` reads left-to-right and thus yields `18` and not the expected `14`!

A.1.4 Cascading Messages

Multiple messages can be sent to the same receiver with `;`.

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi.
```

The message `new` is sent to `OrderedCollection` which results in a new collection to which three `add:` messages are sent with different arguments. The value of the whole message cascade is the value of the last message sent (here, the symbol `#ghi`). This example is the equivalent of first assigning the new collection to a temporary variable and sending three separate `add:` messages:

```
| newCollection |
newCollection := OrderedCollection new.
newCollection add: #abc.
newCollection add: #def.
newCollection add: #ghi.
```

To return the original receiver of the message cascade (*i.e.*, the collection) instead of the last result (*i.e.*, `#ghi`), the `yourself` message is used:

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi;
  yourself.
```

A.1.5 Blocks

Blocks are objects containing code that is executed on demand, (anonymous functions or closures). They are the basis for control structures like conditionals and loops.

```
2 = 2
  ifTrue: [ Error signal: 'Help' ].
```

The first example sends the message `ifTrue:` to the boolean `true` (computed from `2 = 2`) with a block as argument. Because the boolean is `true`, the block is executed and an exception is signaled.

```
#('Hello World' $!)  
do: [ :e | Transcript show: e ]
```

The next example sends the message `do:` to an array. This evaluates the block once for each element, passing it via the `e` parameter. As a result, `Hello World!` is printed.

A.1.6 Methods

Methods are first-class objects in PHARO and can be inspected and modified on the fly. Methods are created by saving expressions in the PHARO development environment. Typically methods are printed with a special first line indicating the class the method is installed on and the name or selector it is given.

```
Array >> helpMethod  
2 = 2  
ifTrue: [ Error signal: 'Help' ].
```

This example would denote a simple method with a unary selector on the `Array` class. This method could be invoked by evaluating `Array new helpMethod`.

Certain methods are marked with a pragma to use predefined primitives from the VM. These are used for expressions that cannot be expressed in PHARO. For instance the `basicNew` which allocates new objects uses the primitive number 70:

```
Behavior >> basicNew  
"Answer a new instance of this class"  
<primitive: 70>  
OutOfMemory signal.
```


Curriculum Vitae

Personal Information

Name: Camillo Bruni
Date of Birth: September 28, 1985
Place of Birth: Bern, Switzerland
Nationality: Swiss

Education

2011-2014: **Ph.D. in Computer Science**
RMod
INRIA, Lille - Nord Europe, France
<http://rmod.inria.fr/>

2008-2011: **Master of Science in Computer Science**
Thesis Title: *Optimizing Pinocchio*
University of Bern, Switzerland
<http://unibe.ch/>