Extending a Java Virtual Machine to Dynamic Object-oriented Languages

Tobias Pape, Arian Treffer, Robert Hirschfeld, Michael Haupt

Technische Berichte Nr. 82

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 82

Tobias Pape | Arian Treffer | Robert Hirschfeld | Michael Haupt

Extending a Java Virtual Machine to Dynamic Object-oriented Languages

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6743/
URN urn:nbn:de:kobv:517-opus-67438
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-67438

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-266-7

Abstract

There are two common approaches to implement a virtual machine (VM) for a dynamic object-oriented language. On the one hand, it can be implemented in a C-like language for best performance and maximum control over the resulting executable. On the other hand, it can be implemented in a language such as Java that allows for higher-level abstractions. These abstractions, such as proper object-oriented modularization, automatic memory management, or interfaces, are missing in C-like languages but they can simplify the implementation of prevalent but complex concepts in VMs, such as garbage collectors (GCs) or just-in-time compilers (JITs). Yet, the implementation of a dynamic object-oriented language in Java eventually results in two VMs on top of each other (double stack), which impedes performance.

For statically typed languages, the Maxine VM solves this problem; it is written in Java but can be executed without a Java virtual machine (JVM). However, it is currently not possible to execute dynamic object-oriented languages in Maxine.

This work presents an approach to bringing object models and execution models of dynamic object-oriented languages to the Maxine VM and the application of this approach to Squeak / Smalltalk. The representation of objects in and the execution of dynamic object-oriented languages pose certain challenges to the Maxine VM that lacks certain variation points necessary to enable an effortless and straightforward implementation of dynamic object-oriented languages' execution models. The implementation of Squeak / Smalltalk in Maxine as a feasibility study is to unveil such missing variation points.

Contents

1. Introduction **11**
1.1. Contributions 12
1.2. Outline 12

2. Maxine and Dynamic Object-oriented Languages **14**
2.1. Introduction to Maxine 14
2.2. Challenges for Dynamic Languages on Maxine 17

3. Representing Squeak Objects in Memory **20**
3.1. Object Models 20
3.2. Squeak Layout 24
3.3. Mapping Squeak to Maxine 26
3.4. Tagged Integers 32

4. Mapping Execution Models to Maxine **34**
4.1. Execution Models and Virtual Machines 34
4.2. Interaction with Object Models 41
4.3. Implementation Process Concept 44
4.4. Built-in Behavior 47
4.5. Stack Access 48

5. Objects in SqueakMaxine **51**
5.1. Configuration with Schemes 51
5.2. The Squeak Layout Approach 57
5.3. The Maxine Layout Approach 65
5.4. Tagged Integers 74

6. Execution in SqueakMaxine **77**
6.1. Architecture of SqueakMaxine 77
6.2. Runtime Object Access 78
6.3. Bytecode Compilation 81

6.4. Garbage Collection in the Presence of Executing Code 91
6.5. Implementing Built-in Behavior 92
6.6. Runtime Execution Information and Call Stacks 101

7. Variation Points of a Virtual Machine Framework 110
7.1. From Configurability to Extensibility 111
7.2. Heap and Garbage Collector 114
7.3. Layout 116
7.4. Other Subsystems 120

8. Maxine Variation Points: An Evaluation 122
8.1. Instruction Set Abstractions 122
8.2. The Template-based Just-In-Time Compiler 124
8.3. Re-usability of Maxine's Compilation Infrastructure 126
8.4. Implementing Built-in Behavior 128
8.5. Stack Manipulation 129
8.6. Applicability to Other Dynamic Languages 130

9. Related Work 133
9.1. Virtual Machines in Higher-level Languages and Metacircular Virtual Machines 133
9.2. Virtual Machine Product Lines, Platforms, and Frameworks 134
9.3. Java Virtual Machines 134
9.4. Squeak Virtual Machines 135
9.5. Dynamic Languages on Java Virtual Machines 136

10. Summary and Future Work 137

A. Key to Figures in Chapters 3 and 5 145

B. Extended Interfaces 147

C. Squeak Essential Primitives 151

D. SqueakMaxine Primitive Implementation Overview 154

E. Key for Virtual Machine Structure Diagram 156

F. Code Examples for S1X 157

List of Figures

3.1. The Java Object Model 21
3.2. The Squeak Object Model 23
3.3. The Squeak Layout 24
3.4. Compiled Method Layout 26
3.5. The Maxine Actor Hierarchy 27
3.6. Relations of the Dynamic Hub 28
3.7. The OHM Layout Hierarchy 29
3.8. The OHM and HOM Layouts 29
3.9. Linking Instance and Class Object in Maxine 30
3.10. The YOHM Layout Hierarchy 32

4.1. Process virtual machine building blocks. 42
4.2. Object model and execution model in a VM 43

5.1. GC Step on the Semi-space Heap 56
5.2. Implementations of the Squeak Layout 61
5.3. The Squeak Class Actor Hierarchy 67
5.4. Compiled Method Layout Conversion 71
5.5. Allocating a Cell for a Squeak Object 71
5.6. Squeak Object Converted into the YOHM layout 73

6.1. Maxine Execution Model Implementation Architecture 78
6.2. SqueakMaxine Execution Model Implementation 79
6.3. Default Java compilation broker behavior 82
6.4. T1X templates overview 84
6.5. Illustration of native code after compilation by T1X. 85
6.6. T1X classes overview. 86
6.7. Squeak compilation broker behavior 88
6.8. T1X frame layout 102
6.9. Married contexts 105

6.10. S1X frame layout 106

7.1. Parallel Environments 111
7.2. Integrated Environments 112
7.3. Garbage Strategy 114

8.1. Factorization of the template-based JIT compilers. 125
8.2. Template compiler proposal 126

List of Tables

5.1. GeneralLayout Methods Implemented for the Squeak Layout . . 59
5.2. SpecificLayout Methods Implemented for the Squeak Layout . . . 60
5.3. The Image Loading Process 69

6.1. Differences between bytecode and primitive behavior 96
6.2. Implementation levels of primitives in SqueakMaxine 100

8.1. Possible locations of bytecode functionality implementations . . . 125

B.1. Additions to Scheme Interfaces 147

C.1. Essential Squeak Primitives 151

D.1. Primitive implementation status in SqueakMaxine 154

List of Listings

4.1. Source of `Object>>at:` 41

5.1. Origin to Cell Conversion in the HOM layout 53
5.2. The OHM Array Layout Implementation 54
5.3. Origin to Cell for the Squeak Layout 58
5.4. ImageIO Example 61
5.5. ImageAccess Example 62
5.6. ImageLoader Example 62
5.7. Printing a Stack Trace 63
5.8. YOHM Tuple Layout Changes 66
5.9. Finding the Behavior Class 70
5.10. SqueakObject#hashCode() 75
5.11. Tagged References in Java Code 75

6.1. `T1X` parts relevant for the `IADD` bytecode. 83
6.2. `S1X` parts relevant for the `PRIM_EQUIVALENT` bytecode. 89
6.3. `SqueakTemplateSource`, Excerpt 90
6.4. Native Methods in `java.lang.Object` 94
6.5. `java.lang.Object` intrinsics' definitions 95
6.6. `S1X` parts relevant for the Squeak primitive 110. 98
6.7. Squeak primitive failure handling 99
6.8. Example use of **`thisContext`** 103
6.9. `ContextPart>>pc` source code 107
6.10. `ContextPart>>pc` bytecodes 107
6.11. Prefixed push instance variable 107
6.12. Runtime part for `variableInContext()` 108
6.13. Visitor to ensure that the caller frame is married 109

7.1. Scanning an Object During Garbage Collection 115
7.2. Object Scanning Deferred to Layout 116

7.3. Visit References 118

F.1. Squeak/Smalltalk source of `Behavior>>new` as of Squeak 4.3. 157
F.2. Squeak bytecode of `Behavior>>new` as of Squeak 4.3. 157
F.3. `S1X` generated native code for `Behavior>>new`. 158

List of Abbreviations

API	application programming interface
AST	abstract syntax tree
CLOS	Common Lisp object system
CRI	compiler–runtime interface
FFI	foreign function interface
GC	garbage collector
HOM	header-origin-mixed
IDE	integrated development environment
JSON	JavaScript Object Notation
JDK	Java development kit
JIT	just-in-time compiler
JLS	Java language specification [15]
JVM	Java virtual machine
JVMS	Java virtual machine specification [25]
JNI	Java native interface
JRE	Java runtime environment
JSR	Java specification request
MLVM	multi-language virtual machine
MOP	metaobject protocol
OHM	origin-header-mixed
OO	object-oriented
OOP	object-oriented programming
PIC	polymorphic inline cache
TLAB	thread-local allocation buffer
UML	unified modeling language
VM	virtual machine
YOHM	origin-header-dummy-mixed
IR	intermediate representation
LIR	lower-level intermediate representation
XIR	compiler and runtime independent intermediate representation

1. Introduction

The traditional way of implementing dynamically typed, object-oriented languages is to write a virtual machine (VM) in a language like C or C++ [6]. Also, statically typed, object-oriented languages such Java or C# are available as C / C++-based VM implementations. Nevertheless, writing VMs in languages without higher-level abstractions — such as proper object-oriented modularization, automatic memory management, or interfaces — has an impact on their maintainability, especially since VMs typically include features that are rather complex to express, e.g., automatic memory management with garbage collector (GC), threading implementations, or just-in-time compilers (JITs). Hence, dynamic object-oriented languages are also written in languages like Java that provide higher-level abstractions to simplify their implementation [6]. Consequently, this typically results in one VM, for the dynamic object-oriented language, running on top of another VM, e.g., the Java virtual machine (JVM) (double stack).

Like dynamic object-oriented languages, statically typed languages can also benefit from the advantages of higher-level–language VM implementations [12]. For instance, Maxine [26] implements a JVM in Java. Maxine has a strong focus on maintainability and tries to leverage the features of Java, such as the automatic memory management or the elaborate annotations system. Maxine can be bootstrapped to machine code and, subsequently, be run without another JVM as host. Therefore, a possible loss of performance due to a double stack is avoided.

For dynamic object-oriented languages implemented in Java and running on top of a JVM, this double stack actually exist. Moreover, running on top of Java bytecode poses a semantic mismatch for dynamic languages. The Java bytecode and JVMs are designed and optimized for the statically typed nature of Java and do not anticipate the dynamic features of dynamic object-oriented languages such as late binding, the notion of "everything is an object" [22], or changing an object's structure at runtime. While approaches that alleviate these problem exist, e.g., a bytecode designed especially for dynamic languages [39], they do not meet the full extent of the semantics of dynamic object-oriented languages; a JVM's assumptions about, e.g., object layout or object access are bound to the static nature of Java. This can affect the performance of dynamic object-oriented languages on JVMs.

We propose to run a dynamic object-oriented language *within* a JVM to circumvent this downsides, i.e., the language implementation is treated as part of the VM and does not target Java bytecodes. This should be feasible with Maxine, as several dynamic object-oriented languages already exist in Java implementations that could be re-used and extended to run inside Maxine. However, to facilitate this on the Maxine part, variation points within Maxine itself are necessary. That way, Maxine should become a *virtual machine framework for dynamic object-oriented languages*.

1.1. Contributions

The aim of this work is to identify missing variation points in Maxine. These would be necessary to enable dynamic object-oriented languages being implement in Maxine with reasonable effort. Squeak/Smalltalk serve as a feasibility study. This work is concerned with how to *execute* behavior of dynamic object-oriented languages within Maxine.

The contributions of this work are as follows:

- the *object model* of Squeak and its representation in memory, and a maping to a layout closer to what Maxine natively supports.
- *execution models* for Java and Squeak/Smalltalk and the conceptual process of mapping execution models to implementations in Maxine;
- an *implementation* of Squeak's object and execution model in Maxine;
- the identification of *variation points* in Maxine that are neccesary to make Maxine a VM framework for dynamic object-oriented languages.

1.2. Outline

The following chapter 2 gives an overview of Maxine and its relation to dynamic object-oriented languages. Then, chapter 3 presents a definiton of *object models* and discusses the representation of objects in memory. Accordingly, chapter 4 defines what an execution model constitutes in the context of dynamic object-oriented languages and VMs, and explains, how to map such models to Maxine.

Then, chapter 5 presents the implementation of object models and chapter 6 reports on the implementation of Squeak in Maxine, carried out as an application of the tasks given in chapter 6. Chapter 8 evaluates the findings of this application. After the presentation of related work in chapter 9, this work closes with a summary and conclusion in chapter 10.

2. Maxine and Dynamic Object-oriented Languages

This chapter presents an overview of Maxine and its concepts with special attention drawn to the question how execution happens in Maxine. The challenges that dynamic object-oriented languages might pose for an implementation in Maxine follow subsequently.

2.1. Introduction to Maxine

Maxine is a VM for the Java language and is written in Java. It is developed at Oracle Labs (former Sun Labs) and the project members understand Maxine as a "Platform for Virtual Machine Research" [42].

The Maxine VM aims to provide a VM that is usable in place of the standard JVM [25]. It shares the same format for code and uses the same class library. Unlike the standard JVM, Maxine does not provide interpreted execution of Java bytecode but only just-in-time compilation.

2.1.1. Maxine as a Java Superset

Implementing a virtual machine (VM) usually requires access to low-level data structures, notably, means to address memory. One of the core parts of a VM is the memory management, which is dependent on such structures. Moreover, execution infrastructure, like JITs, need these structures, e.g., to provide entry points to compiled methods or to compile memory accessing functionality. The Java language deliberately abstracts from such low-level structures, providing automatic memory management instead. To enable the implementation of the Maxine VM in Java, Maxine augments the standard Java class library by a machine word data type hierarchy. These types are handled in a special way by Maxine's compilers, yielding machine code suitable for pointer manipulation or memory addressing purposes. In this sense, the Maxine platform offers a superset of the Java language.

2.1.2. Bootstrapping

The Maxine VM is written in Java, nevertheless, it does not need another JVM to run. This is achieved by *bootstrapping* the VM from within a JVM. Currently, only the Java HotSpot VM is supported for this task. The process of bootstrapping consists of the following steps:

- Start the JVM with the bootstrapping program.
- Load the Maxine modules that are necessary to run the Maxine VM. This includes GC, JITs, and startup sequences, among others. The respectively chosen implementations depend on the configuration of schemes and brokers as described below.
- Create Maxine's internal object representation of the just loaded modules, their classes, and methods.
- Compile all methods that are reachable from the loaded modules.
- *The previous two steps are repeated with all new entities and methods encountered during compilation steps until a pre-defined boundary is encountered, e.g., a certain call depth or certain methods of the class library.*
- Write all collected objects and compiled methods to an *image file*.

Together with a small, platform-dependent loader program written in C, the image now constitutes a Maxine VM. Executing the loader program just loads the image file into memory and jumps into the startup scheme that was specified at bootstrap time. From this point, compiled Java code is executed natively; a host JVM is no longer necessary.

Depending on the bounds for the image, the Maxine VM possibly has to load other Java code — excluding the actual Java application code. Hence, the Java JITs always are part of the Maxine image and a Maxine VM always will act as a JVM, regardless of what language, object model, or execution model is actually implemented.

2.1.3. Variability through Schemes

One of the main goals of the Maxine project is to provide compile-time configuration of the Maxine VM by employing standard Java features. Maxine uses *schemes* for that purpose [51]. A scheme is a Java interface defining how the Maxine VM interacts with the described component, e.g., the heap management/GC. During boot image generation, concrete implementations for schemes can be selected, e.g., a copying GC or a mark-and-sweep GC.

Employing variability through schemes allows Maxine to act as a software product line [8]; certain coarse grain functional aspects can be exchanged prior to generating the product, in this case a VM, more accurately, a specific Maxine VM. This specific VM consists of a small loader and the Maxine boot image.

Currently, the following schemes exist in Maxine: the *Layout scheme* defines the layout of Maxine VM objects in the memory; the *Heap scheme* defines the layout of the VM's heap and the kind of GC to use; the *Reference scheme* defines the access to objects for manipulation; the *Monitor scheme* defines how synchronization and locking is implemented; and the *Run scheme* defines the start-up procedure right after the VM initialization.

2.1.4. Pluggability through Brokers

Besides compile-time configuration, Maxine has a runtime variability facility, called *broker*s. A broker comes into effect, when several components provide a similar feature, but the decision which to choose is runtime bound. The currently sole instance of this technique is the choice of the JIT for a method. While there are several JITs available, it is infeasible to determine at compile-time of the Maxine VM, which one should be used throughout the built VM. This is due to the characteristics the different JITs have; be it optimization or compatibility to the original bytecode, some methods call for a different JIT than others. Hence, all necessary compilers are included at compile-time and a broker is used to determine the JIT necessary for a certain method in a certain state. This broker is called the *Compilation broker*.

There is only one instance of brokers currently present in Maxine, but this concept can be extended as needed.

2.1.5. Execution in Maxine

Maxine's primary task as VM clearly is executing Java programs. Such programs typically are pre-compiled as Java bytecode with which VMs have to deal. Unlike the standard HotSpot™ based JVM, Maxine does not have any interpreter but JITs. These transform bytecode into native machine code.

Although Maxine has more than one JIT, the execution behavior of Maxine is the same for them. When a Java method is to be executed, Maxine looks up the method's internal representation. Depending on the method's state, the bytecode of the method is compiled using one of Maxine's JITs, if necessary. Either way, Maxine uses the method's native machine code to continue execution.

Maxine has a few hand-crafted machine code parts to facilitate calls between optimized and non-optimized methods and a handful of C-written functions that perform platform dependent operations, such as initial memory allocation. Apart from these functions, all functionality within Maxine is provided by JIT-compiled Java methods.

2.2. Challenges for Dynamic Languages on Maxine

Implementing dynamic object-oriented languages in Maxine might face certain difficulties that are either due to the properties of the languages themselves or due to the current state of Maxine regarding its design choices and their implementation. While the former kind of difficulties can be identified by analyzing the languages, the latter needs a more in-depth investigation of Maxine's implementation to identify difficulties or, more appropriately, missing variation points.

2.2.1. Relevant Characteristics of Dynamic Object-oriented Languages

Of the well-known or popular programming languages, dynamic object-oriented languages feature a combination of characteristics that are interesting for implementation in Maxine. This includes characteristics that are both present in, at least several, dynamic object-oriented languages and Maxine, facilitating an implementation in the first place. However, this also includes characteristics that are not currently present in Maxine, hence, implementing these can enrich Maxine.

Similar characteristics Certain characteristics of dynamic object-oriented languages match features Maxine provides for its Java implementation.

VM BASED Many dynamic object-oriented languages are already run on VMs, e.g., Self, Lua, or JavaScript. Some designed that way, e.g., Smalltalk, some just happen to be implemented for or as VMs but were not designed to, e.g., JavaScript. Nevertheless, it is not unnatural for dynamic object-oriented languages to be run on VMs, which suits Maxine.

Memory management Most dynamic object-oriented languages use automatic memory management with the requirement for garbage collection. The challenge here is to integrate Maxine's garbage collection with that of the dynamic object-oriented language, even more so, to allow for garbage collection during the execution of methods of the dynamic object-oriented language. As indicated in section 2.1.2, a Maxine vm will always be able to operate a as jvm internally, regardless of which language is actually implemented. Hence, methods of both the dynamic object-oriented language and Java may exist in a Maxine vm for the dynamic object-oriented language, both bound to interact with the gc.

Intermediate representations Dynamic object-oriented languages such as Smalltalk or Lua, among others, have a distinct intermediate representation their source code is compiled to. The languages vms ever only see this intermediate representation, e.g., bytecode. This is similar to Java and, hence, Maxine can work with bytecode representation. Note, however, that other dynamic object-oriented languages like Ruby or Python do not have such an intermediate representation *exposed*; their implementations are expected to deal with source code directly.

Object model Every dynamic object-oriented language has a distinct object model. Such an object model can be different from the Java object model implemented in Maxine.

Dissimilar characteristics Some characteristics of dynamic object-oriented languages do not match with features present in Maxine and, thus, pose a particular challenge for dynamic object-oriented languages on Maxine.

Lookup The method lookup for dynamic object-oriented languages is normally unlike the method lookup for Java. Features like genuine polymorphism, e.g., in Smalltalk, or object-specific behavior, e.g., in JavaScript or Self are challenges to be considered as there are no concepts present in Maxine for them, yet.

Closures Most dynamic object-oriented languages provide means to create and invoke closures, be it blocks in Smalltalk, Self, or Ruby; λ-expressions in Python or Ruby; or anonymous functions in JavaScript. Maxine has no abstractions for closures, as Java does not have a true closure mechanism, yet. However, upcoming releases of Java are said to include syntax and semantics for closures. Hence, tackling closures in the process of bringing dynamic object-oriented languages to Maxine can be beneficial to an implementation of closures for Java later on.

Reflection and live systems Most dynamic object-oriented languages provide means to introspect and intercede running programs. Inspecting and creating classes and methods is possible in, e. g., Python and Ruby, and also in Smalltalk, where this functionality is vital, as reflection is the only means of introducing new behavior to the system. Self, on the other hand, provides a sophisticated *mirror* system that, even more than in Smalltalk, poses the means of program introspection, intercession, and maintenance. While in Java reflection is limited to access to class and object fields and invocation of methods, the creation and alteration of shape and behavior like in most dynamic object-oriented languages is not possible. However, the means for behavior and state creation are understandably present in Maxine to support running programs in the first place. Yet, altering shape and behavior of existing internal object is not supported, yet, and has to be considered to allow for *live* systems on Maxine.

2.2.2. Maxine as a Virtual Machine Platform

At the time of writing, Maxine is a platform for Java virtual machines. As pointed out, the guest language of any Maxine VM is Java in its bytecode form. The platform aspect is provided by the two configuration mechanisms, schemes and brokers, that make it possible to produce differently laid out VMs from the same code base, comparable to software product lines. This makes Maxine a Java VM platform.

Taking the idea of a VM platform further, Maxine as a platform for dynamic object-oriented languages seems reasonable. Are the current means of configuration fit for the implementation of other languages than Java? Moreover, is it possible to extend Maxine to a dynamic object-oriented language implementation *framework*? The difference here is, whether Maxine is just used to implement a dynamic object-oriented language VM or whether it is capable of *facilitating* a dynamic object-oriented language VM implementation by providing enough variation points and inversion-of-control points. xIn the context of execution, this is evident when looking at possible implementations of, e. g., execution engines: A platform should provide a common interface for all execution engine implementations, a framework should provide generic execution engines that can be specialized for a given language. This holds for execution engine–garbage collection, too, among others.

3. Representing Squeak Objects in Memory

Even though many modern programming languages provide the familiar concept of objects, the understanding of what an object is and can do differs between languages. This is relevant when representing objects in memory; the implementation should be efficient, yet reflect the languages semantics of objects.

This chapter begins with a definition of the term "object model" and a comparison of the object models of Java and Squeak. Then, we show how object models can be mapped to memory, analyze how this was achieved by Squeak for Smalltalk and Maxine for Java, and propose an adaption of the latter to better suit the needs of this project.

3.1. Object Models

Generally, the term "object model" can be used to describe different concepts. While it can be used for any model that is composed of objects, in the context of object-oriented programming languages and their VMs, the term usually refers to how objects are modeled in the language. Nevertheless, there is no widely accepted definition of this term. Snyder [45] used it as a terminology to propose a generic object concept, but focused on language-agnostic client-server systems. The definition used in this work is based on a survey from 1997 [30], which was reduced to relevant aspects.

3.1.1. Definition

In the context of OOP languages, an *object model* defines the properties and characteristics of objects of a language. All object-oriented programming (OOP) languages have a concept of *relations* between objects, internal *state* of an object, *behavior* that works on an object's state, and means to *induce* the behavior. A language's object model defines how relations and state are

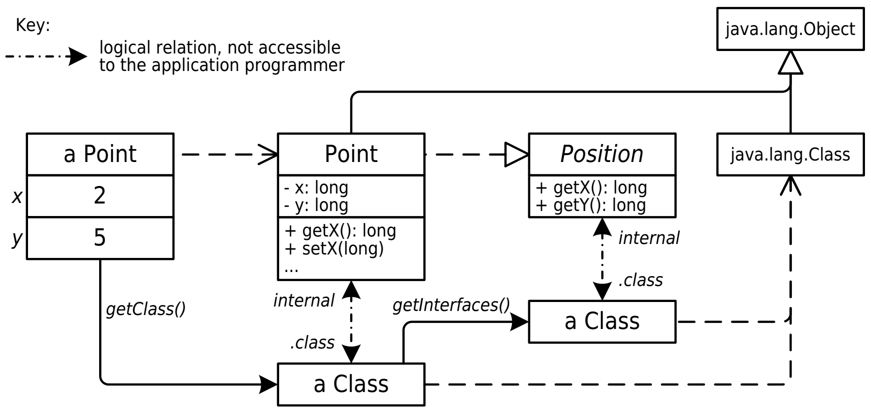


Figure 3.1.: The representation of a point in Java. Language entities such as classes and interfaces are accessible only through reflection objects. The diagram syntax is explained in Appendix A.

conceptually represented, e.g., as indexed or named fields; where meta information is stored, e.g., in the object or its class; and which operations are allowed at runtime, e.g., only reading from a constant field or adding new fields to single objects. Furthermore, it describes where the behavioral elements of an object can be found, e.g., individual to an object, organized in a class or globally available. Finally, the object model defines which meta objects the language provides; this usually includes some reflection application programming interface (API).

Maxine already provides a fully working object model implementation for Java. A comparison with the model that is to be realized can show which parts of the implementation might be reusable and reveal potential incompatibilities.

3.1.2. The Java and Squeak Object Models

Java is a statically typed, class-based object-oriented language. As such, classes define fields that store state and relations, and methods that implement behavior. Both can be declared to be either class-bound (static), which means they are globally unique and can be accessed directly; or instance-bound, in which case an instance of the defining class is required for access or invocation. An object is an instance of a class and contains only its state, i.e., its fields' values. As a consequence, the class is required to actually access fields or invoke methods.

Except for `java.lang.Object`, which is the inheritance root, every class extends exactly one class. A subclass may add fields and introduce new or override existing methods. Furthermore, a class can implement an arbitrary number of interfaces. An interface defines a set of method signature without providing an implementation. They provide flexibility, as Java ensures type-safety and supports no other means of multiple inheritance. Aside from classes, Java also has primitive types for characters, integers and floating point numbers. Language entities, such as classes, fields, or methods, have no direct representation and can only be accessed through the reflection API.

The example in figure 3.1 shows how the point $(2, 5)$ can be represented as a Java object. The point is instance of the `Point` class, which implements the `Position` interface; both are represented by a class object, which can be accessed by navigating from the point object or via their literals, `Point.class` and `Position.class`, respectively.

Smalltalk uses an object model that is based on very few simple design principles [46, 14]:

1. Everything is an object.
2. Objects communicate via message passing.
3. Classes describe in terms of state (instance variables) and behavior (methods) the objects they generate.
4. When an object receives a message, the corresponding method is looked up in the class (and super class) of the receiver.
5. Methods are public.
6. Instance variables are private.
7. Classes inherit via single inheritance.

Smalltalk was the first pure object-oriented language, where "everything is an object." As such, there are no primitive types; every variable contains a reference. Principles 3, 4 and 7 describe the same concept of classes that Java uses: an object is but a data storage and its class is required to interpret the data or to invoke behavior. Behavior can be shared through single inheritance only. As Smalltalk is dynamically typed, there is no need for interfaces.

The dynamic aspect is strengthened further by the second rule, which indicates that even though a message has a name, it is at the object's very own discretion to decide which method is invoked in response. Of course, the default behavior is to look up a method of that name (cf. principle 4), but

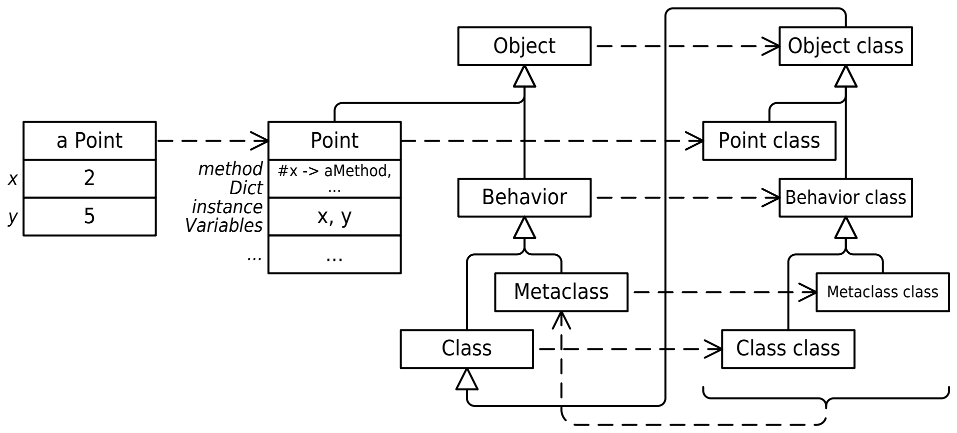


Figure 3.2.: The representation of a point in Squeak. Classes are objects as well and defined in Squeak's meta-hierarchy.

if no such method is found, a callback method is invoked that receives the message's name and arguments as parameters. This way, any message can be received and processed, without the necessity to implement a certain interface; however, by default the object may chose to signal that the message was not understood.

As messages contain no information about the sender, all methods are public (principle 5). On the other hand, fields, as defined by principle 6, are always private, which automatically provides type-safety for field accesses.

Another difference to Java caused by the "everything is an object" principle is that classes and methods are affected as well. Shown in figure 3.2 is the same point-example as above, but for Squeak. On the left hand side, the point object is shown, storing the field values, along with its class, defining fields and methods. This time, however, the class is an object as well and instance of another class. The right hand side shows a selection of Squeak's meta objects, which are necessary to implement this concept. All class objects are instances of subclasses of `Class`, which again are all direct instances of `Metaclass`.

This means that some Squeak objects are not instances of a class, but of a meta class. Thus, whenever a Squeak object's class is mentioned in this work, this actually refers the object's *behavior*, regardless of whether it is a meta or regular class.

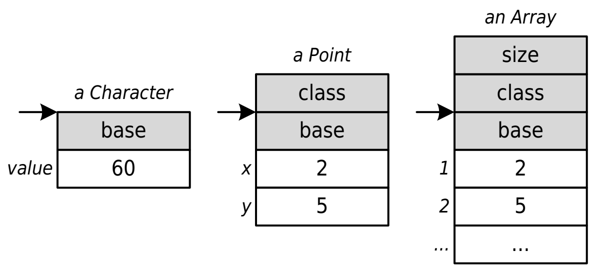


Figure 3.3.: The Squeak layout. Each Squeak object cell contains at least a base header. Depending on the object's properties, additional header words can be prepended.

3.1.3. Object Layout: The Physical Representation

As part of an object model implementation, the *object layout* defines how objects are stored in memory, which includes the representation of state and meta information. A full implementation further has to provide means to locate the state and behavior of an object, a way to allocate new objects, and, for many languages, a garbage collector.

There may be different implementations of the same object model. For instance, Bacon, Fink, and Grove [4] have done four implementations of the Java object model for the Jikes Research VM. Note that they use term "object model" for concrete memory layouts of an abstract object model.

3.2. Squeak Layout

In Squeak, every object is represented as a piece of memory containing the state, preceded by one to three header words. These additional words are used by the Squeak VM to obtain meta information.

3.2.1. Header Words

Given in figure 3.3 is an example for every possible header layout. Each block represents one word of memory; text outside of blocks shows implicit information. Header words are highlighted in gray. The arrow shows the *origin* of each object, which is the address that is used when referencing an object. In Squeak, an object's origin is always the address of its base header. The *base header* word can contain everything the VM needs to know about an object.

As with every header word, the lowest two bits indicate the total number of header words of the object. Other bits can store the size of the object cell in bytes, the compact class id, garbage collector flags and the cell's layout kind, which will be explained in a moment.

Squeak maintains an array of the most frequently used classes, the *compact classes array*. A class's *compact class ID* is its index in this array, or zero. Instances of compact classes have this index stored in the base header; for all other objects an additional *class header* is required, which points to the object's class. For very large objects, the base header bits reserved for cell size are not sufficient. In this case, a third header word is required. Just as the base header, the *size header* contains the object size in bytes, *not* including the two optional header words.

3.2.2. Nine Layout Kinds

Squeak distinguishes between nine types of objects with different layouts. The type is chosen by the class, but is, for performance reasons, also represented by a 4-bit number in the base header. These layouts are as follows:

NO FIELDS is used by Objects with no state beyond their identity. The most prominent examples are `nil`, `true` and `false`.

FIXED FIELDS objects have a fixed number of named fields, which are defined by the class.

INDEXABLE FIELDS indicates a simple object array. The array length can be calculated from the cell size.

HYBRID begins with a fixed, named part, followed by indexable fields.

WEAK HYBRID works just like hybrid, but the GC treats references as weak.

WORD ARRAY is used for arrays of 32-bit words. Unlike with the indexed fields layout, the object cannot contain references.

LONG ARRAY is the 64-bit variant of word array.

BYTE ARRAY indicates an array of bytes. There are four values representing the byte array layout, with the lower two bits indicating the number of alignment bytes.

COMPILED METHOD are byte arrays with a more complex internal structure, as shown in figure 3.4. Its first content word acts as an additional header and encodes, amongst other technical information, the number of literals in the method. The literals are references; the remaining bytes are relevant for the methods execution and contain bytecode.

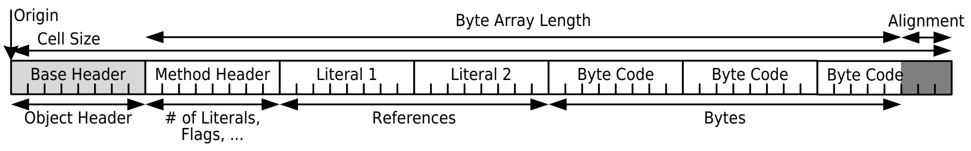


Figure 3.4.: Compiled method layout. A `CompiledMethod`, shown here in 64-bit layout, contains a reference and a byte array part. The length of the reference part is encoded in the method header field.

3.2.3. The Squeak Heap

All Squeak objects are allocated on the heap, which includes all classes and even the integrated development environment (IDE). At any point, the current VM state can be saved, by storing the entire heap in a file, preceded by some header data. The image header also references the main entry point of the application: the *special objects array*. At predefined indices, it contains important objects, such as `nil`, `true` and `false`, the class dictionary, and the active process. The latter stores a stack and the last active execution context, from which the execution of the image can be continued. To load an image, the file is mapped into memory. With a single iteration all object references are adjusted to fit the new memory address range. After that, the execution can be continued exactly where it was stopped.

A direct implementation of this layout in Maxine could support image loading and saving easily. Such an implementation was started and will be presented in section 5.2; it was, however, not completed for reasons that will be discussed there as well.

3.3. Mapping Squeak to Maxine

To allow for a better reuse of Maxine's infrastructure of class and method management, a layout was needed that is closer to Maxine's default layout, while still being flexible enough to support Squeak specific features not present in Java. This section will first explain how Maxine natively handles objects and then propose an extension to this layout that is able support objects with Squeak-specific properties.

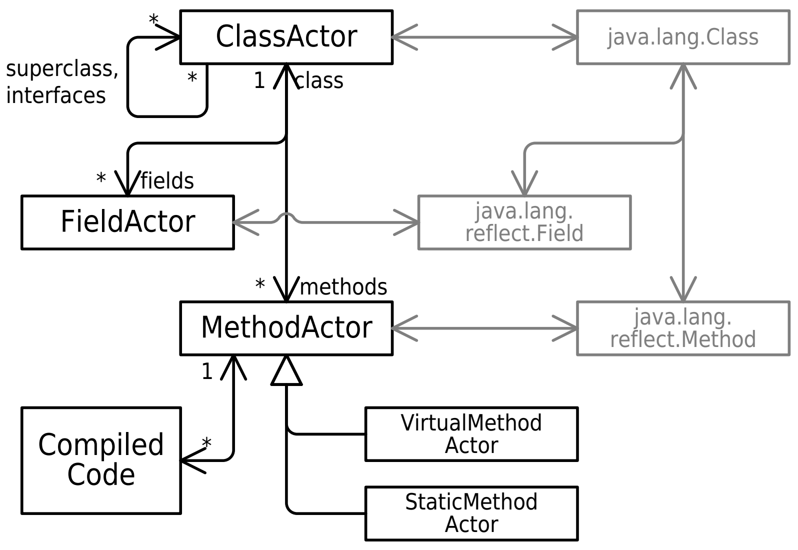


Figure 3.5.: The Maxine actor hierarchy. Maxine's actors represent language entities and can be accessed by the client program through the reflection API.

3.3.1. Hubs and Actors

Unlike classes in Squeak, Java classes are no objects. Thus, the object model does not specify how language entities are to be represented by the VM.

In Maxine, every class and interface, method, and field is represented by an *actor*. Actors are first-class objects that "act" for their corresponding Java entities by carrying out the underlying actions of Java instructions. The client application can access actors only indirectly via the Java reflection API. A part of the actor architecture and how it interacts with Java's reflection API is shown in figure 3.5.

Associated to each class are two more helper objects: the static and the dynamic hub. As figure 3.6 shows, every object references its class's dynamic hub in a header field and thereby indirectly knows its class. One important task of the hub is to maintain the *vTable*. The vTable is a hash table referencing the entry point of each of the class's instance or static methods, depending on the hub type, and allows to resolve virtual method calls in constant time. A dynamic hub further maintains the *iTable* and *mTable*, to resolve interface method calls in constant time, too.

The details of these tables' functionality are not relevant here; nevertheless, they add an interesting property to hub objects. Usually, all Java objects are

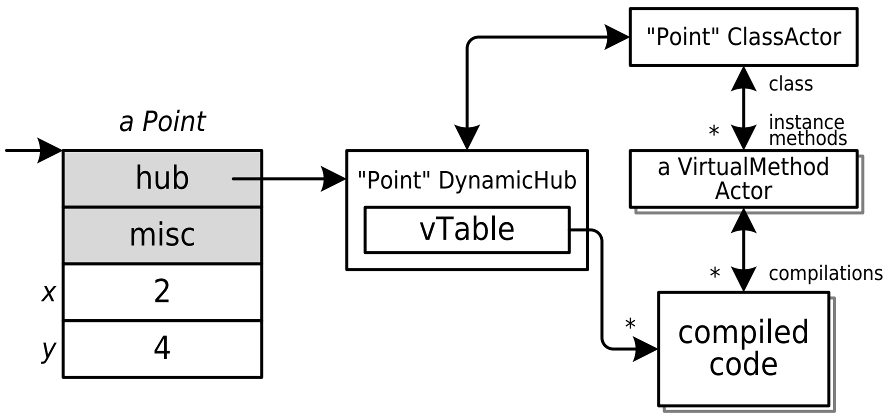


Figure 3.6.: Relations of the dynamic hub. Every object references a dynamic hub, which is used to resolve method calls and to perform other meta operations, such as type checks.

either tuples of fixed size determined by the class or arrays of variable length. Hubs, however, are hybrids and begin with fields, e. g., for the class actor and the layout, followed by a reference array taken up by the three tables.

3.3.2. The OHM Layout

Just as Squeak, Maxine knows different layout kinds. In Maxine, a layout implementation has to provide a general layout, which can read and write header fields without knowing details about an object. Maxine by default uses the following header fields:

HUB Each object stores a reference to its class's dynamic hub in the *hub header*.

MISC Every Java object can be used as a synchronization monitor, as defined by Hoare [18], with a corresponding wait queue. The *misc header* encodes details of the locking state and stores the identity hash code, obtainable via `System.identityHashCode()`.

LENGTH For variable sized objects, the *length header* stores the number of contained elements.

Additionally, a Maxine layout implementation has three specific layouts: a tuple layout for instances of regular classes, a hybrid layout for hubs, and an array layout. The array layout is further specialized into a general array layout, agnostic of the element type, a reference array layout for all object arrays, and one array layout for each primitive type.

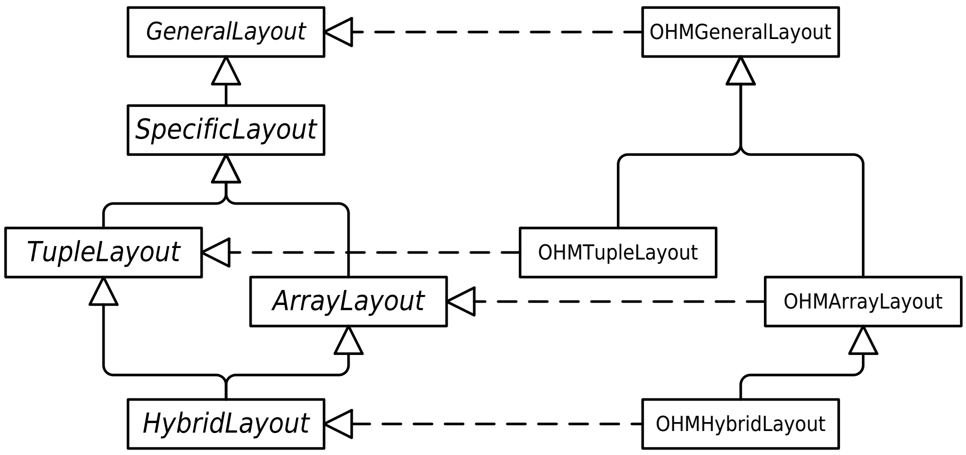


Figure 3.7.: The OHM layout Hierarchy. The OHM layout provides a simple implementation of the layout interface hierarchy and is instantiate by the OHM layout scheme.

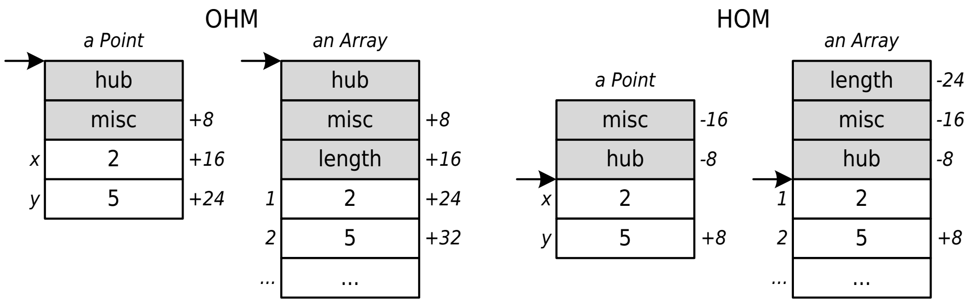


Figure 3.8.: The OHM and HOM layouts. Maxine's two layout implementations differ in the placement of the origin and the ordering of the header fields.

This leads to a hierarchy of interfaces, as shown in figure 3.7 along with the default implementation, the origin-header-mixed (OHM) layout. In figure 3.8, it is compared with a second layout implementation, the header-origin-mixed (HOM) layout. By placing the origin behind the header, the HOM layout provides easier access to array elements and is closer to Squeak. The OHM layout allows for a quicker access of the hub.

Field access work the same for both layouts: the corresponding field actor knows the field's offset from the origin, which is then compiled directly into assembler code before a method is executed the first time.

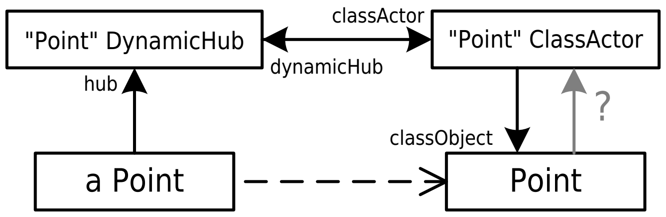


Figure 3.9.: Linking instance and class object in maxine. The hub and class actor can be used to create a bridge between an instance and its class object.

At the time SqueakMaxine was developed, full support was guaranteed only for the OHM layout, whereas the HOM layout was considered deprecated. Thus, any extension of existing layout mechanisms would have to be based on the OHM layout.

3.3.3. A Compromise

To present Squeak objects in a way Maxine is already familiar with, the different kinds of Squeak layouts have to be mapped to the existing OHM layout kinds. As a precondition, the header fields have to be converted appropriately: the length header value for variable sized objects can be calculated from the object and element size, the misc header is Maxine specific and is initialized just as it is for every regular Java object, and the hub has to handle everything else that was stored in the class and base header in Squeak, i.e., the specific layout, the class and the fixed tuple size.

Consequently, an appropriate hub is required for each Squeak class. Since hubs are created and initialized by their class actors, this already points the way to how Squeak classes can be introduced to Maxine: every Squeak class must have its own class actor. This class actor has to mirror the Squeak class. Thus, it is initialized with information about all of the classes' fields and methods, which leads to the creation of the according field and method actors as well. Finally, as the hub has to be used to find an object's Squeak class and is, on the other hand, required to create new instances of a class, a bidirectional association between class actor and Squeak class object has to be established (cf. figure 3.9).

Here, it comes as an advantage that the implementation of Java classes is not defined by the object model. This way, it is easy to subclass the existing class actor implementation to introduce a field which references the Squeak class object. On the Squeak side, things are a bit more complicated: it is not possible to add a field as this could violate assumptions in the VM or in metaprograms, especially since this field would not reference a Squeak object.

Java requires that all classes inherit from `java.lang.Object`, an assumption that thus is made in several parts of the VM. In consequence, Squeak's inheritance root `ProtoObject` needs a Java super class. To provide some basic type-safety when handling Squeak objects in the VM, the abstract class `SqueakObject` was created. This allows for variables in the VM code which can contain Squeak objects only and enables the identification of Squeak objects with `instanceof`.

3.3.4. Introducing New Layout Kinds

Now that the hub is in place, other parts of the VM, such as the GC, can use it to get the object's specific layout and access the object's content. However, as mentioned before, appropriate specific layouts have to be provided. For SqueakMaxine, the origin-header-dummy-mixed (YOHM) layout was introduced. To test the flexibility and extensibility of the OHM layout, the YOHM layout was to reuse as much existing code as possible, being modified only where it had to support features not present in Java.

For six of Squeak's nine layout kinds, the mapping was rather simple. The no-fields and fixed-fields layouts directly match Maxine's tuple layout: the object size is the same for all instances and defined by the class. Further, Squeak's reference, word, long and byte array layouts can be mapped to their respective array implementations easily. For Squeak's two hybrid layout kinds, it was beneficial that Maxine already supported hybrid objects in the form of hubs. However, in hubs the array part contains integers only, whereas Squeak hybrids contain references. Since both versions of the hybrid layout are required, support for reference hybrids was introduced as a modified copy of the original variant. Compiled methods, although superficially equivalent to byte arrays, require special support for the integrated literal references arrays, and thus require a dedicated layout as well. This leads to the layout hierarchy shown in figure 3.10.

Until now, the existing specific layouts were not modified at all, but there is one requirement of Squeak that is not yet supported. In Maxine, all hybrid classes extend `com.sun.max.vm.object.Hybrid`, which has no fields and is a direct subclass of `java.lang.Object`. In Squeak, however, it is possible for every tuple class to have a hybrid subclass. The resulting layout switch, adding the length header to the object cell, would invalidate the field offsets

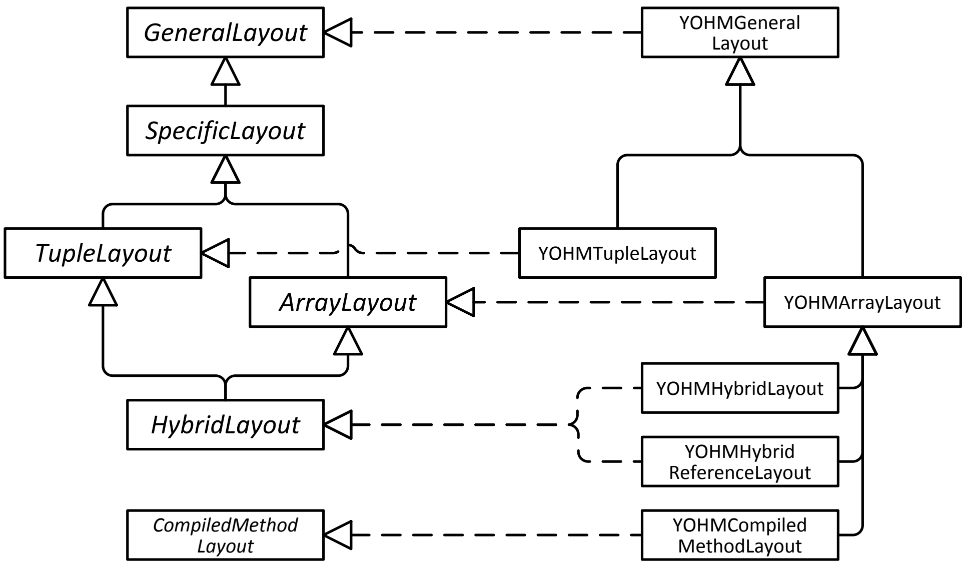


Figure 3.10.: The YOHM layout hierarchy. The YOHM layout implements the regular layout hierarchy, but also provides additional classes.

of the super class's fields. When code compiled for the super class tries to access the first field at +16, it would now find the length header (cf. figure 3.8).

To avoid this problem, an additional header word was introduced for tuples. The dummy header field, represented by the additional letter in the newly named "YOHM layout", originally served no purpose other than aligning the tuple fields. Later, it found its application in storing the class actor reference for Squeak class objects, implementing the missing link shown in figure 3.9. Therefore, it is also known as the class actor header field. Using the adjusted tuple layout, the YOHM layout can now map every kind of Squeak object into memory.

3.4. Tagged Integers

Squeak's object model poses a particular challenge with respect to implementing integers.

First of all, an integer object somehow has to represent a value. It cannot, like Java's `java.lang.Integer`, have a field that stores its value as a primitive. In Squeak, this value would have to be an object, i.e., the integer object itself.

Secondly, integer operations are a frequent task in computation, much more than floating point operations (except maybe in physical simulations and similar tasks). If every operation with an integer result would require an object instantiation, the execution would be slow and inefficient, once for the additional memory allocations and once because of the increased frequency of garbage collector runs. The only workaround seems to be a huge cache of reusable integer instances, which would require a lot of memory and still could cover only integers in a relatively small range.

Both problems can be avoided by exploiting a property of memory architecture: A pointer value specifies a memory offset in bytes, even though it is not possible to read or write individual bytes, but only entire words. Thus, pointers have to be *word aligned*, which means their lowest two or three bits are always zero. As objects are stored in memory, the same constraint of course also holds true for object references. Thus, it is possible to *tag* a reference, i.e., mark it as somehow special, by setting one or more of these bits to one. Squeak uses this mechanism to implement tagged integers. If the lowest bit of a reference is set, it indicates that the other 31 bits do not reference a valid memory address, but directly encode the numeric value of the object. Thus, every time the class of an object is required, e.g., to resolve a late-bound method invocation, the lowest bit of the object reference has to be checked. If it is set, the object is an instance of `SmallInteger`; otherwise, the address can be accessed to read the object's header fields.

To perform mathematical operations on small integers, the values have to be left-shifted by one first. Afterwards, it has to be checked whether the result still fits into 31 bits. Larger integers and floating point numbers can be implemented as full objects now, using multiple small integers to store their data. Bringing tagged references to Maxine would create a small overhead every time the hub is accessed. Nevertheless, it still provides an efficient solution to the problems described above and probably should be included in any Squeak implementation.

⁂

Both layout solutions presented in this chapter are a possible way to represent Squeak objects in Maxine. As an addition, a new way to handle references to special objects was introduced. An implementation of these three ideas will help to evaluate the decoupling of Maxine's subsystems, and show the overall flexibility of the code.

4. Mapping Execution Models to Maxine

This chapter introduces the notion of execution models and gives the execution models of Java and Squeak, the latter in anticipation of the implementation to be described in chapter 6. This is followed by the execution model parts to be minded when implementing an execution model in Maxine. For each such part, the conceptual mapping process is given.

4.1. Execution Models and Virtual Machines

To be able to describe the mapping of an execution model to Maxine, first, a working definition for *execution model* is given, followed by the execution models of Java and Squeak in the sense of that definition.

4.1.1. Working Definition in the Context of Virtual Machines

Considering process virtual machines for object-oriented programming languages, an *execution model* of a language denotes the semantics of method execution and method lookup defined by *built-in behavior* and by *intermediate representations (IRS)*, *formal or informal semantics*, or the combination of the aforementioned, whichever is regarded *canonical* for the respective language. This excludes implementations details.

For clarification, the language's semantics — whether formal, informal, or merely given by a reference implementation — always contain *language-level semantics* and may contain *IR-level semantics*. When a canonical version for the latter exists for a language, it is considered to be sufficient for the languages execution model.

Built-in Behavior All operations that cannot be expressed in the language and are crucial for the language to operate properly or are canonically expected to be provided by a VM form the *built-in behavior* of a language and are part of its execution model.

EXAMPLE: SQUEAK OBJECT CREATION In Squeak, the method `#basicNew:` represents the primitive with the number 71 and deals with the allocation of memory in the course of object creation. The method cannot be expressed in Squeak itself but is vital for Squeak to work properly and, hence, belongs to Squeak's execution model.

EXAMPLE: JAVA LOCKING Every Java object can be used as a condition variable for locking purposes. The methods `notify()`, `notifyAll()`, and `wait()` provide this locking behavior. However, this cannot be expressed in Java for arbitrary objects, e. g., a plain, filed-less object that has no state that used to store in the information necessary for the locking behavior. Hence, these three methods contribute to Java's execution model.

Intermediate Representations (IRs) Processed, non-code representations of methods of the language form *intermediate representations* of those methods. This includes, but is not limited to, bytecode representations. When regarded as canonical, intermediate representations might be part of the language's execution model. If such an intermediate representation specifies a certain machine model, the machine model itself becomes part of the execution model.

EXAMPLE: SQUEAK BYTECODE In Squeak, the intermediate representation for methods is the Squeak bytecode which is emitted by Squeak's runtime compiler. For any VM that should be regarded a Squeak VM it is canonically expected to support Squeak bytecodes, the Squeak bytecode is part of Squeak's execution model.

EXAMPLE: JAVA BYTECODE Java classes and methods are compiled to class files that contain, among other data, bytecodes for Java methods. This is normally done by the `javac` compiler. Any VM that should be regarded a Java VM has to adhere to the Java virtual machine specification [25] (JVMS), and hence is canonically expected to support Java bytecodes; the Java bytecode is part of Java's execution model.

EXAMPLE: PYTHON BYTECODE The CPython implementation of Python (canonical for at least Python 1 and 2) generates Python bytecodes for the methods it interprets. However, this fact is regarded as implementation detail and is not canonically expected to be supported in all Python implementations. Hence, the Python bytecodes are *not* part of Python's execution model.

EXAMPLE: RUBY AST REPRESENTATION The MRI implementation of Ruby (canonical until version 1.8) uses an abstract syntax tree (AST) representation for the interpreted Ruby methods prior to method execution. This is an implementation detail and not canonically expected of all Ruby implementations, hence not part of the execution model of Ruby. In fact, the AST representation has vanished in Ruby version 1.9.

Semantics Specified formal or informal semantics for execution of methods in the language may be part of a languages execution model, if canonically expected. Informal semantics include language standards.

EXAMPLE: SMALLTALK FORMAL SEMANTICS Although semantics for Smalltalk exist [52], they are not canonically expected to be followed in Smalltalk implementations. Hence, these semantics are not part of Smalltalk's execution model, and neither Squeak / Smalltalk's.

EXAMPLE: SCHEME FORMAL SEMANTICS While only mediately object-oriented via implementations of the Common Lisp object system (CLOS), the execution of Scheme is specified formally in the quasi-standard for Scheme R^5 by Adams et al. [1, § 7.2] The formal semantic hence contributes to Scheme's execution model.

EXAMPLE: PYTHON OPERATOR OVERLOADING The fact that the expression `a + b` results in the method `a.__add__(other)` being called is part of the semantics of Python method execution.[1] This is not implementation specific and canonically expected of all Python implementations and, hence, part of Python's execution model.

Canonically Expected The phrase 'canonically expected' used above describes the circumstance that a certain execution specification is regarded as required to be fulfilled by implementations of the language. This can be either convention, specified standard, or both.

EXAMPLE: SQUEAK The relevant specifications for Squeak are the set of primitives an the semantics of the Squeak bytecode and are followed by the major existing Squeak implementations like the Squeak interpreter VM, the

[1]Note that while the Python Reference treats this circumstance as part of the data model [37], by the definition given here, it belongs to the execution model.

Cog VM, or the RoarVM, to name a few. Although formal semantics for Smalltalk exist that might be applicable for Squeak [52], none of the mentioned implementations claim to fulfill the semantics. Hence, primitives, i.e., built-in behavior, and bytecode, i.e., intermediate representation, belong to the Squeak execution model, as opposed to semantics.

EXAMPLE: PYTHON The Python execution model comprises built-in behavior [36] and, additionally, semantics of certain operations [37]. Existing bytecode specifications [35], however, explicitly are implementation details, hence not part of the execution model.

EXAMPLE: JAVASCRIPT The execution of JavaScript is defined in the respective ECMA standard [11].

EXAMPLE: JAVA For Java, especially the VM part, a specification exists [25], which among other VM aspects defines bytecode meanings and, hence, at least parts of the execution model. Java VMs are expected to adhere to the specification.

Language-level and IR-level Semantics If the language has however specified intermediate representation-level semantics in addition to its language-level semantics, the intermediate representation-level semantics contribute to the languages execution model; the language-level semantics do not need to. If there is no IR-level semantics, the language-level semantics contribute to the execution model.

EXAMPLE: SQUEAK/SMALLTALK As pointed out, the Squeak bytecode is part of Squeak's execution model. This does not withstand that the *language* Squeak/Smalltalk has semantics, e.g., as specified by the Blue book [14] or the 1998 standard [3]. However, from the execution model's point of view, the bytecode is sufficient as the language-level semantics does not add any value to the execution model; a Squeak/Smalltalk VM normally does not even "see" semantic elements of the language, i.e., source code in this case.

EXAMPLE: JAVA Java as a whole is defined by two specifications, the Java language specification [15] (JLS), e.g., in its third version [15] and the Java virtual machine specification [25] (JVMS), e.g., in its first edition [25] (Adaptions to these exist for nearly each respective Java version). The distinction made for language-level and IR-level semantics is quite evident here: the JLS contains specifications at the language level, the JVMS at VM level. The latter also contains a full specification of the Java bytecode. The semantics as specified by the JVMS suffice to execute Java, therefore, only the JVMS is necessary for the execution model.

EXAMPLE: PYTHON Python is neither formally nor informally specified using IR-level semantics. As already pointed out, the existence of bytecodes for CPython is an implementation detail. That is why the language-level semantics of Python contribute to its execution model.

4.1.2. The Java Execution Model

The Java execution model consists of a specified *intermediate representation* and *built-in behavior*. For Java, both a *language-level semantics* — specified by the JLS — and an *IR-level semantics* — specified in the JVMS — exist.

INTERMEDIATE REPRESENTATION The intermediate representation of Java is its bytecode set as specified by the JLS.

BUILT-IN BEHAVIOR The minimum set of indispensable `native` methods in the Java runtime environment (JRE) constitute the built-in behavior of Java.

N.B.: It is evident, that the execution model of Java changes whenever either of the specifications changes or the native methods of the JRE change their semantics between releases. Hence, there are different Java execution models; this work assumes the execution model of Java 1.6 as specified by the JLS in the third edition, the JVMS in the second edition — both with respective updates from Java specification request (JSR) 270[2] — , and the Java development kit (JDK) in version 1.6.0_26.

Java intermediate representation Java's IR is its bytecode or instruction set [25]. Method bytecodes are fed to the JVM via class files, also specified in the JVMS. Java class files contain information on classes, their relationships to other classes, and the classes methods. These methods, in turn, consist of some information, like class membership and access control, and the code of the method expressed as a list of said bytecodes.

Java Built-in behavior The Java execution behavior is not only defined by the semantics of the Java bytecodes but also by the functionality of the classes of Java's standard library, or more specifically, the standard library's core. This core is shipped as Java archive (commonly `rt.jar`), a combination of

[2]`http://jcp.org/en/jsr/detail?id=270`

the standard Java packages and classes. Those classes mostly contain bytecode compiled methods. However, several methods are not expressed by bytecode but are rather `native` methods, especially those that cannot be expressed with Java bytecode. Depending on the platform, between ca. 1,250 methods (Mac OS X) and ca. 1,900 methods (Linux) in the core Java archive are `native` methods. We assume that several methods that are `native` in the core Java archive are actually expressible by Java bytecode but are implemented natively, e.g., for performance reasons. Omitting methods that are in theory expressible as Java bytecode, only the indispensable `native` methods remain, forming the built-in behavior of Java.

4.1.3. The Squeak / Smalltalk Execution Model

The Squeak / Smalltalk execution model consists of an *intermediate representation* and *built-in behavior*. The specification, influenced by the Blue Book [14] and the standard [3], is chiefly given by the reference implementation as of the official `VMMaker`[3] package.

INTERMEDIATE REPRESENTATION The intermediate representation of Squeak / Smalltalk is its bytecode set.

BUILT-IN BEHAVIOR All "Essential" Squeak / Smalltalk primitives constitute the built-in behavior of Squeak / Smalltalk.

N.B.: The `VMMaker` used as reference for SqueakMaxine is `VMMaker` 4.2.2, and for selected parts of context handling `VMMaker.oscog` 4.6.0 (Cog).

Squeak / Smalltalk intermediate representation Squeak / Smalltalk's IR is its bytecode set. It is a stack-based, object centric bytecode set that is documented in the Blue Book [14], but vastly augmented in the `VMMaker` Squeak VM generation application. There is no standard body defining the actual set of bytecodes for Squeak / Smalltalk; rather, community decisions or pioneering VM implementations have impact on what bytecode set is *canonical*.

Squeak / Smalltalk bytecodes are contained in instances of `CompiledMethod` that are generated by the Squeak / Smalltalk compiler in a Squeak image.

Smalltalk Build-in Behavior Smalltalk and, hence, Squeak / Smalltalk has the notion of *primitives* that may provide functionally that is not — or not

[3] `http://wiki.squeak.org/squeak/2105`, source accessible via `http://source.squeak.org/VMMaker.html`

as fast — possible in Smalltalk on its own. Most of the Smalltalk primitives are described in the Smalltalk implementation description by Goldberg and Robson (called the *Blue Book*) [14], with Squeak providing an extended, altered set of the original Smalltalk primitives. However, not all primitives contribute to the built-in behavior of Squeak / Smalltalk as many of the primitives exist mainly for performance reasons. Squeak / Smalltalk knows of a maximum of 575 primitives by index, of which about 220–230, depending on version and VM implementation, are actually used in Squeak. The remaining indices are either unused, unimplemented, or used for special *quick return* optimizations. In the unused and unimplemented case, the primitive invocation just fails. In the special quick return case, methods that, e.g., just implement the return of the receiver or an instance variable, are given certain unused primitive indices for optimization purposed. Hence the difference between the number of indices and the number of actually implemented primitives. The Blue Book specifies 128 primitives, all primitive indices higher than 128 denote Squeak / Smalltalk specific primitives in the above sense, i.e., including quick return indices.

A number of primitives have a special comment in the methods using them, stating that this primitive be essential, and these primitives constitute the Squeak built-in behavior. An example of an essential primitive is `Object>>at:` as given in listing 4.1. Only the "Essential." marking in line 4 identifies this method's primitive as being part of the built-in behavior. The compiler-directive in line 7 specifies which primitive to invoke when `#at:` is sent to an object. The method body specifies the code to execute in case the primitive fails or is unimplemented.

As of Squeak version 4.3, 65 unique primitives are marked as being essential in the Squeak image. However, the reliability of the essential set is hence not well tested; this definition of the built-in behavior of Squeak / Smalltalk remains therefore disputable in this regard.

Since the same indexed primitive can be invoked by different methods, more than 65 methods bear the "Essential." marking. This makes a total of 89 marked methods that map to the 65 essential primitives. For reference, please refer to table C.1 in Appendix C.

Besides primitives that are identified by index, there are primitives that can be called by readable identifiers, called *named primitives*. Much more named primitives than indexed ones exist, yet they all do not contribute to the built-in behavior of Squeak / Smalltalk.

```
Object>>at: index
  "Primitive. Assumes receiver is indexable. Answer the value of an
  indexable element in the receiver. Fail if the argument index is not an
  Integer or is out of bounds. Essential. See Object documentation
  whatIsAPrimitive."

  <primitive: 60>
  index isInteger ifTrue:
    [self class isVariable
      ifTrue: [self errorSubscriptBounds: index]
      ifFalse: [self errorNotIndexable]].

  index isNumber
    ifTrue: [↑self at: index asInteger]
    ifFalse: [self errorNonIntegerIndex]
```

Listing 4.1: Source of `Object>>at:` in a Squeak 4.3 image. The method is declared to try the indexed primitive 60 and provides functionality in case the primitive fails. Note the statement "Essential." in the documenting comment.

4.2. Interaction with Object Models

Throughout this work, we use a model of VM that divides a VM into two main parts, an object model implementation and an execution model implementation, based on the assumption that both, an object model and an execution model exist for the language provided by the VM. This twofold model is an abstraction of the process VM structure as given by Smith and Nair [44].

Seeing that it is possible to divide a dynamic object-oriented language into its object model and its execution model, making the same distinction for a VM implementation for such a dynamic object-oriented language seems reasonable. Hence the abstraction of the Smith and Nair-based process VM building blocks as in figure 4.1 into an *object model implementation* and an *execution model implementation* as depicted in figure 4.2.

Note that the term *guest (language)*, as used in figure 4.1, denotes the language that is provided by the VM, while the term *host (language)* would refer to the language that is used to implement the VM itself.

Object model implementations are most important for objects of the VM's guest language. As pointed out, in Maxine as a meta-circular VM, such an implementation has an impact on the whole VM's memory management. Objects of the host language are subject to the same mechanisms as object of the guest language. They are hard to tell apart. Actually, in Maxine, they are just the same.

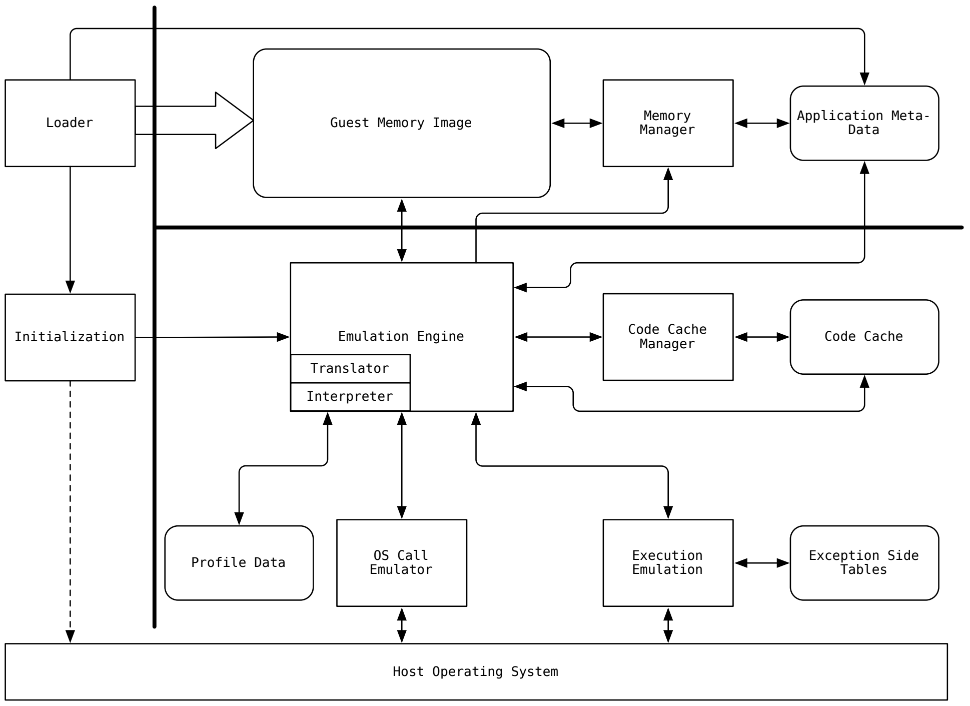


Figure 4.1.: Process virtual machine building blocks [16, slides 4, 18], based on [44], with model boundaries added. Key: see Appendix E.

Storing Non-guest Information For some aspects of VM implementations, it is desirable to store more information in a guest object than visible to the guest language. Such *non-guest information* has to be accessible from the VM itself but not from the guest language. An example for this is locking in Java. Every object can act as a condition variable, however, the state information required to achieve this is to be hidden from Java. There is no field in the Java `Object` class exposing this locking state, it is internal to the VM. Another example would be the class of an object for many class based dynamic object-oriented languages and Java.

An approach is to have a lookup table mapping from an object to the requested information, but this is cumbersome and quite inefficient. Object tables as in the original Smalltalk-80 are an example of this approach, extended to the lookup of the object itself and not just properties for it.

A more typical approach for storing non-guest information is the use of header fields. Considering the locking example, the information required to

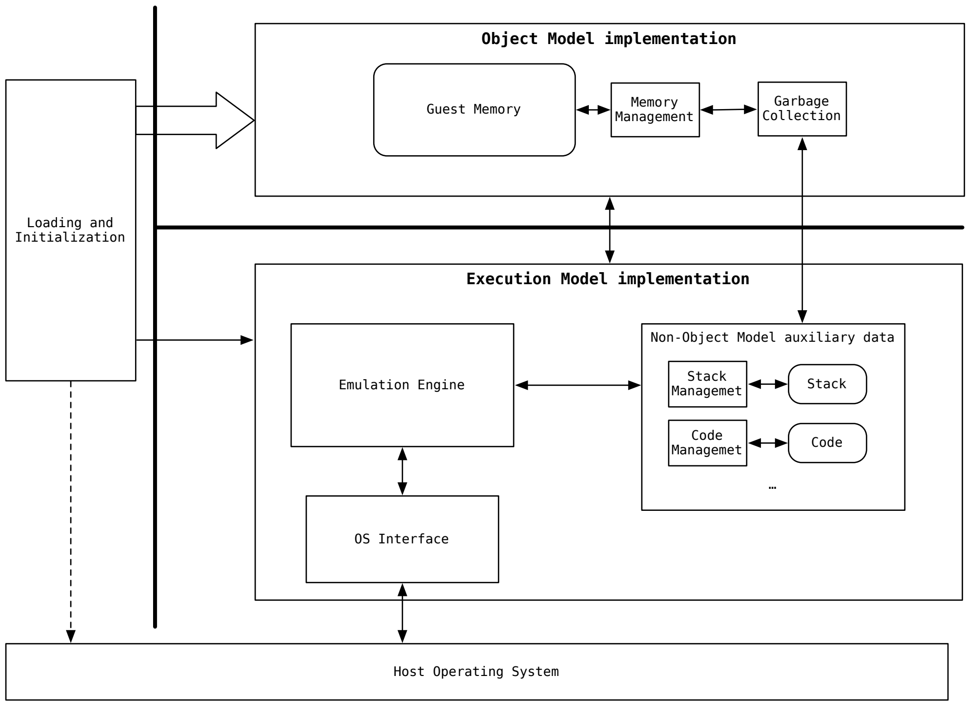


Figure 4.2.: Object model and execution model in a VM, abstracted from a VM's building blocks (cf. figure 4.1). Key: see Appendix E.

determine whether an object is currently a condition variable it is encoded as a header field in Maxine.

A third approach is to use specially tagged values in an object, such that the VM can store associations to internal data structures. This approach is taken by the Squeak stack VM in order to maintain a link between a guest object representing a context and its accompanying actual stack frame. This approach is feasible if there is no other way left; it has an impact on the understandability of the object model implementation as possibly many checks are necessary when accessing the respective values in an object.

However, depending on the language implemented, it might be necessary to support more non-guest information associated with an object.

N.B.: The question of how or where to store non-guest information might seem to be of interest for object models and object model implementations only. Despite that, the implementation of an execution model might require non-guest information to be present in certain guest object, which cannot be anticipated by an object model implementation. In general, this is always the case, when data that belongs to the "Non-Object Model auxiliary data" like in figure 4.2 must be associated with guest object. Object model implementation an execution model implementation hence have to overlap in this very respect.

Maxine and Non-guest Information Maxine's approach to provide non-guest information in guest objects is the use of header fields. Providing more non-guest information than currently done for Java, hence, requires the addition of header fields to an object. That additional header field is used for non-guest information in SqueakMaxine's guest class objects. Using that non-guest information, it is possible to draw the connection between the guest class object and the SqueakMaxine class actor object that represents the respective class.

For any implementation of non-guest information in Maxine, it is hence subject to the object model information to provide such storage, in order to be used by an execution model implementation.

4.3. Implementation Process Concept

The heart of an execution model implementation in VMs for dynamic object-oriented languages is the part that actually carries out execution. This part, called *emulation engine* in figure 4.2, is responsible for transforming guest language instructions in such a way that they have actual effect in the host environment. The most prominent techniques for such a transformation are interpreters and just-in-time compilers. *Interpreters* directly take instructions in form of an IR from guest language code and carries out these instructions in the host environment in place of the guest code. Think of interpreters as proxies between semantics of the guest code and semantics of the host environment. On the other hand, *just-in-time compilers* do not themselves take actions in place of the guest code but rather map guest code semantics to host environment semantics. They do not act as man-in-the-middle as interpreters do.

Maxine deliberately only contains JITs and no interpreters, as opposed to, e.g., the standard JVM that has both an interpreter and a JIT. Hence, the process of implementing intermediate representations in Maxine can safely omit interpreter-related considerations and focus on JITs.

4.3.1. Transformation Requirements

The assumptions Maxine makes about JITs and the native code they are expected to produce, are actually modest.

DATA STRUCTURE ASSUMPTIONS Guest code that is contiguous and forms a semantic grouping, e. g., methods or functions, has to be represented as *actor*. Maxine has *method actors* for all entities that represent executable components. A method actor is expected to be able to access the IR of the method it is acting for.

GUEST CODE AND NATIVE CODE RELATION If guest code has a native code representation, i.e., if it has successfully been compiled, a method actor is expected to keep track of and hold onto that native code. A method actor is expected to expose whether or not it holds onto native code.

TRANSFORMATION PROCESS INDUCTION The transformation process from intermediate representation to native code of the host environment is initiated by a Maxine *compilation broker* and does not need to be initiated manually. Once a method actor, representing certain guest code, is to be executed the broker checks whether the method actor bears native code, and if not, requests the method actor to be compiled.
The broker has to have knowledge about the method actors it is making decisions for.

TRANSFORMATION FUNCTIONS The compilation broker of Maxine will call registered *transformation functions* for method actors without native code. The transformation functions are the entry-points for JITs and expected to take no more information than the method actor to generate native code for.

NATIVE CODE REQUIREMENTS Maxine provides abstractions for entities of native code, called *target methods*. These should contain native machine code for their respective target, i.e., processor type. The code a JIT generates and puts into a target method may provide different entry points when called from methods with a different calling convention. *Adapters* then can convert between these conventions; Maxine already provides adapters for calls between Java baseline-compiled methods, Java optimized-compiled methods, pure C functions, and Java native interface (JNI) function calls.

To subsume, the interface between Maxine and a guest language is a *compilation broker that knows of a method actor specialized for the guest language*.

In addition to these requirements, Maxine has the notion of the *nature* of a target method. A *baseline* method is not expected to execute fast but to be able to execute every control flow path of the method it was compiled for.

Moreover, if the IR of that method was bytecode, a baseline method should also provide a mapping between the original bytecode and locations in the generated machine code, especially for debugging purposes. An *optimized* method does not have these restrictions but is supposed to execute faster.

4.3.2. Possible Implementation Process

The requirements of Maxine to JITs and native machine code as well as the facilities provided by Maxine to aid the creation of JITs suggest the following modus operandi for implementing them.

First, the entities of the guest language that represent execution should be represented by new Maxine internal *actors*, which can be achieved by subclassing the Maxine `MethodActor`. Note that this holds for anonymous behavior like closures, too, as instantiation of such actors typically happens at runtime. It is necessary that these actors expose their nature as either *baseline* or *optimizing*, and that they contain the guest language's IR. Likewise, *target methods* should be created in the same manner, holding onto the to be created machine code and providing information about how the stack should look like for such a target method. The existing target method classes can serve as blue print here.

Next, the actual JIT is needed. Maxine does not place more restrictions on it than that it generates proper machine code, is invocable like the other JITs in Maxine, and consumes instances of `MethodActor` subclasses. However, Maxine provides utilities for JITs, like re-usable template code, called *Snippets* that can be used for common tasks such as compiling call instructions. An existing JIT such as `T1X` can be used as template here. Eventually, the new JIT should fill a target method with the appropriate native machine code and associate the provided method actor with the new target method.

Then, the existing *compilation broker* should be replaced by an augmented one capable of distinguishing between Maxine's own method actors and the guest language's ones. This broker then should call the new JIT whenever a guest language's method actor is to be compiled, and hand on to the original one in any other case.

Finally, at boot-image-creation time, the new compilation broker can be set as the default broker in the Maxine VM, at which point the compilation infrastructure for the guest language should ready to use.

4.4. Built-in Behavior

In Java and dynamic object-oriented languages, execution semantics are not only defined by bytecode or similar IRs. Certain semantics are provided by *built-in behavior*. Such behavior is language dependent and might effect, e.g., specialized interpretation of certain method calls (cf. section 4.1.1).

4.4.1. Instances of Built-in Behavior

The manifestation of built-in behavior of dynamic object-oriented languages might differ vastly. As an example, consider Smalltalk, where built-in behavior is explicitly requested from the VM by method that have a *primitive* tag. Note that multiple methods in Smalltalk can bear a *primitive* tag. Similarly, in Python, a set of built-in functions exist, e.g., **`map`**`()`, **`reduce`**`()` and **`eval`**`()`. Along with the basic object hierarchy, such built-in behavior in Python is found in its `__builtin__` module, making Pythons built-in behavior explicit even by naming. Contrarily, Ruby's built-in behavior is hard to tell apart from normal behavior a VM is dealing with at IR level. In the reference implementation, some functionality "just happen" to be implemented in C when not expressible in Ruby or when performance is critical.

4.4.2. Maxine and Built-in Behavior

The variety in built-in behavior manifestation demands a quite generic implementation approach. Not yet anticipated in Maxine, there is no general handling of built-in behavior of different languages. For Java, Maxine takes an practical approach. Maxine re-uses the standard JDK, including its (C++ based) implementation of Java's built-in behavior. By means of the standardized Java foreign function interface (FFI), the JNI, Maxine is able to call any **`native`** method of the JDK. However, not all implementations of built-in behavior in the JDK fit Maxine. For methods, that need a different implementation in Maxine that in the standard JDK, Maxine provides means to *substitute* the JDK behavior with Maxine-specific behavior.

Hence, Maxine's way of providing Java's built-in behavior is using the JNI, i.e., a *foreign function interface* (FFI), and *substituting* the default external behavior when necessary.

4.4.3. Providing General Built-in Behavior in Maxine

While not directly facilitated by Maxine, it is possible to derive a general implementation approach.

First, it is necessary to provide the actual behavior in a way Maxine can access it. This means that the functionality required has to be either implemented in Java or accessed via JNI. Considering the former, several locations for such implementation are imaginable. It is possible to provide the functionality as part of *Snippets* — small re-usable code fragments for common, low-level functionality like method lookup — for inclusion of built-in behavior into methods that are compiled at runtime. Alternatively, the functionality can be implemented in *Runtime* classes, that are compiled ate bootstrapping time (cf. section 2.1.2). Invoking functionality implemented this way requires only a little native code calling a method in the runtime class.

Second, there has to be a mapping of the language elements that represent the built-in behavior and the functionality as implemented in Maxine. As pointed out, for the Java implementation in Maxine, this part is done via FFI and substitution. However, other mapping mechanisms are imaginable, e. g., by including checks into the lookup mechanisms to deviate from the default lookup when certain methods are to be executed. This would be a feasible way for, e. g., Python's built-in functions. Another way is to intercept the IR parts that represent invocation, e. g., `invoke...`-bytecodes in Java or `send`-bytecodes in Smalltalk. At that point, the requested built-in behavior can be looked up and inlined directly into the method to be compiled. A third imaginable way is possible, when the language's built-in behavior supports treatment as if the behavior was, e. g., yet another bytecode. An example for this is Smalltalk: its built-in behavior, i.e., primitives, is numbered and only one per method is allowed. Hence, it is possible to view the *primitive number* as the "zeroth" bytecode of a method, treating Smalltalk bytecodes and primitives equally.

4.5. Stack Access

To maintain the reflective properties present in many dynamic object-oriented languages and to support debugging in the first place, it is necessary that implementations of execution models have access to the execution stack.

4.5.1. Stack Access in Dynamic Object-oriented Languages

Dynamic object-oriented languages often include functionality supporting introspection of the execution stack, if not even its modification. The most prominent Language with support for both is Smalltalk with an explicit notion of the execution stack as linked list of *contexts*. This list, and each individual context as well, is both inspectable and modifiable during execution. This allows different execution paths to be induced directly from the language level. Other dynamic object-oriented languages also require stack access, at least to read from it. Consider Python, where stack trace can be inspected at language level. A dedicated library, `traceback`, is available to ease the handling of such stack trace objects, but, obviously, needs access to the execution stack. In the same sense, Lua supports the creation and inspection of tracebacks by means of the `debug.traceback()` function. In Lua versions prior to 5.2, it was even possible to modify the current stack frame by means of the **`getfenv`**`()` / **`setfenv`**`()` functions.

Another functionality requires stack access; closures with non-local returns have to have access to the stack to determine their enclosing lexical context. Language supporting closure with non-local returns include, first and foremost, Smalltalk, Self, and Ruby. Similarly, Pythons `nonlocal` keyword for accessing variables of the same name in the enclosing context needs stack access when implemented, too.

4.5.2. Maxine Stack Walking

The requirements for stack access in Java are modest, akin to that in Python, i.e., stack traces must be possible. Hence, Maxine does not provide more stack access than necessary to Java, which is actually only in two places, `Throwable.getOurStackTrace()` and `Throwable.fillInStackTrace()`. Yet, Maxine internally needs a more powerful access to the stack, especially in certain operations of the object model implementation and within that, mainly in the part of the GC that is responsible for detecting objects within the arguments of currently running functions. These are present on the execution stack. Another internal application of stack access is the deoptimization [23, 19] of currently executed methods, that have a frame in the current execution stack. In the course of deoptimizing such methods, their corresponding stack frames have to be replaced, or at least patched, with a version fitting the new nature of the method.

To provide a uniform approach for both the Maxine-internal and the Java-stack-trace stack access, Maxine provides *stack walkers* with a visitor-based interface. That way, the operation of introspecting the current execution stack is solved generically by visiting all frames of the stack with a given closure. For GC on the one hand and Java stack traces on the other, only the action carried out on each stack frames differs.

Maxine has two kinds of stack walkers. One provides an object-oriented view on the call stack and each frame and is called *stack frame walker*. As it allocates during its operations, it is merely used in Maxine's tools, as the Inspector. The other stack walker does not provide an object-oriented view, but passes the raw memory pointers for the relevant registers, i.e., stack- and frame-pointer and the program counter. This walker is hence called *raw stack frame walker*, and as it does not allocate during its operation, it is used for all stack access currently necessary in Maxine.

4.5.3. Implementation Approaches

Given a way to induce stack access from the language, i.e., by means of an IR node or certain build-in behavior, providing read-only stack access is comparatively easy. A visitor handed to a stack frame walker can collect all information necessary for the aforementioned languages' stack accessing functions. Such information only has to be processed to fit the guest languages needs. This operation is straightforward.

However, providing write access to the execution stack has not been done up to now. Write access in this sense does not mean changing, e.g., locals in a stack frame but changing the stack altogether, e.g., forking the stack by modifying frame- and stack pointers. These operations are not accessible in the default stack walker implementation in Maxine. Other means than the default stack walkers have to be devised when stack modification is necessary.

⁂

Having laid out the conceptual implementation approaches for execution models in Maxine, the next chapter will exemplary apply these approaches to one dynamic object-oriented language.

5. Objects in SqueakMaxine

Before the implementation of the concepts introduced in the last chapter can be discussed, this chapter gives an introduction to Maxine's configuration mechanism and explains the relevant subsystems, especially the layout and heap scheme. Then, the implementation of the two layout approaches is presented. Even though only one was completed to the point that it allows to execute code, important insights to improve Maxine's extensibility were gained from both approaches. Finally, this chapter shows how tagged integers have been realized, and how they were integrated in both layout implementations.

5.1. Configuration with Schemes

As a research VM, Maxine was developed with a strong focus on configurability. Logical parts of the VM are encapsulated in so called *schemes*. A scheme is a "Java interface that defines the interaction between a subsystem and the rest of the VM" [41]. Concrete scheme implementations are selected when the build process is triggered and are instantiated and initialized during the boot image generation.

This design allows for multiple implementations for each aspect of the VM, for instance with different performance characteristics. For this project, schemes were used to introduce the object model of another programming language. Since there was no need to rewrite or change the Java-related functionality of Maxine, new schemes were introduced by sub-classing or copying existing implementations. To given an impression of how VM internals are realized in Maxine, the remainder of this section presents Maxine's existing schemes and explains parts of their implementation, as much as they are relevant for the object model realization.

5.1.1. The Layout Scheme

Most important for the object model is the *layout scheme*, which defines how the content of an object cell has to be interpreted. This is done by providing an implementation of the layout hierarchy, as already shown in figure 3.7. Unlike all other schemes, which are defined as interfaces, `LayoutScheme` is an abstract class. By default, Maxine uses the OHM layout, which already was conceptually explained in section 3.3.2. Here, we show how a layout can be implemented, using the Maxine pointer API.

In listing 5.1.1, the implementations of some methods in the OHM general layout are shown. In lines 3 to 9, the offsets of the header fields are calculated. The hub header is located directly at the origin, the misc header follows by one word. The general layout includes only header fields that are present in every object, which means the length header for arrays and hybrids is not defined here.

```
 1 public class OhmGeneralLayout
 2  extends AbstractLayout implements GeneralLayout {
 3
 4  final int hubOffset = 0;
 5  final int miscOffset;
 6
 7  public OhmGeneralLayout() {
 8   this.miscOffset = hubOffset + Word.size();
 9  }
10
11  public final Reference readHubReference(Accessor a) {
12   return a.readReference(hubOffset);
13  }
14
15  public final void writeHubReference(Accessor a, Reference hub) {
16   a.writeReference(hubOffset, hub);
17  }
18
19  public final Word readMisc(Accessor a) {
20   return a.readWord(miscOffset);
21  }
22
23  public final void writeMisc(Accessor a, Word value) {
24   a.writeWord(miscOffset, value);
25  }
26
27  public final SpecificLayout specificLayout(Accessor a) {
28   return getHub(a).specificLayout;
29  }
30  public final void writeForwardRef(Accessor a, Reference fwdRef) {
31    a.writeReference(hubOffset, fwdRef.marked());
32  }
33
34  public final Reference readForwardRef(Accessor a) {
35    final Reference forwardRef = a.readReference(hubOffset);
36    if (forwardRef.isMarked()) {
37      return forwardRef.unmarked();
38    }
39    // no forward reference has been stored
40    return Reference.zero();
41  }
42 }
```

Accessors to the header fields are defined in lines 11 to 25. The `Accessor` interface, which is implemented by the `Pointer` class, makes these low-level operations easy to implement. Furthermore, the general layout can be used to determine an object's specific layout, which can, for instance, be obtained from its hub (cf. line 28). The method `getHub` is not shown here, it simply converts the result of `readHubReference` into a `Hub`, using an unsafe cast.

Another important layout functionality that cannot be deferred to specific layouts is storing forward references (cf. lines 31–42), specially tagged references indicating that an object was moved by the garbage collector. Using the pointer API, the OHM layout defines that forward references are stored in the hub header field. Since hub references are never tagged, it is always possible to detect whether an object has been forwarded.

```
public Pointer originToCell(Pointer origin) {
  return !isTuple(origin) ?
    origin.plus(arrayLengthOffset) : origin.plus(hubOffset);
}

public Pointer cellToOrigin(Pointer cell) {
  return cell.readWord(0).asAddress().isBitSet(0) ?
    cell.minus(arrayLengthOffset) : cell.minus(hubOffset);
}
```

Listing 5.1: Origin to cell conversion in the HOM layout. In the HOM layout, cell and origin pointers are not identical and have to be converted. The conversion has to regard that arrays and tuples have an additional header field.

Finally, the general layout is responsible for the conversion between cell and origin pointers. As has been shown for the HOM layout (cf. figure 3.8), they are not necessarily the same. The HOM layout's conversion code is given in listing 5.1. For a given object pointer, the cell start can be calculated by determining the object's layout kind and adding an appropriate (negative) offset to the origin. Given a cell pointer, however, the object's layout kind cannot be determined as it is not possible to locate the hub. To distinguish whether the first word is a hub or a length header, for the latter the lowest bit is always set to one. By checking this bit, it is then possible to derive the offset to the origin. In consequence, every time the length header is read or written, it has to be remembered that the actual value is left-shifted by one. For the OHM layout, however, both methods always just return their parameter.

Responsible for implementing different layout kinds are the specific layouts. As an example, listing 5.2 shows how relevant parts of the OHM array layout are implemented. The array layout is instantiated multiple times (cf. the constructor parameter in line 8), once for each primitive type and once for `Reference`. The reference array layout is used for all arrays with an object component type.

```
 1 public class OhmArrayLayout
 2  extends OhmGeneralLayout implements ArrayLayout {
 3
 4  public final int lengthOffset;
 5  public final int headerSize;
 6  public final Kind elementKind;
 7
 8  public OhmArrayLayout(Kind elementKind) {
 9   lengthOffset = miscOffset + Word.size();
10   headerSize = Word.size() * headerFields().length;
11   this.elementKind = elementKind;
12  }
13
14  public boolean isArrayLayout() { return true; }
15  public Layout.Category category() { return Category.ARRAY; }
16
17  public HeaderField[] headerFields() {
18   return new HeaderField[]
19    {HeaderField.HUB, HeaderField.MISC, HeaderField.LENGTH};
20  }
21
22  public int originDisplacement() { return headerSize; }
23
24  public boolean getBoolean(Accessor accessor, int index) {
25   return accessor.getBoolean(originDisplacement(), index); }
26  public byte getByte(Accessor accessor, int index) {
27   return accessor.getByte(originDisplacement(), index); }
28 public char getChar(Accessor accessor, int index) {
29  return accessor.getChar(originDisplacement(), index); }
30 ...
31 public void setBoolean(Accessor accessor, int index, boolean b) {
32  accessor.setBoolean(originDisplacement(), index, b); }
33 public void setByte(Accessor accessor, int index, byte b) {
34  accessor.setByte(originDisplacement(), index, b); }
35 public void setChar(Accessor accessor, int index, char c) {
36  accessor.setChar(originDisplacement(), index, c); }
37 ...
38 }
```

Listing 5.2: The OHM array layout implementation.

Once an object's specific layout is known, `is(Tuple|Array|Hybrid)Layout` and `category` can be used to determine the object's layout kind. The specific layout further defines which header fields it uses (cf. lines 21–23) and provides accessors for header fields that are not handled by the general layout (not shown in the example). Finally, the specific layout allows to access the contents of the object. For the array layout, as shown in lines 28 to 39, there is one pair of access methods for every component type, which allow to read and write one element of the array; for the tuple layout, a field actor is used to identify the field that should be accessed. Using these functions, it is possible for the VM to inspect and modify objects efficiently, without the overhead of Java's reflection API.

5.1.2. The Heap Scheme

A second scheme responsible for handling objects is the *heap scheme*. Unlike the layout scheme, which manages the contents of single objects, the heap scheme manages a set of objects, regarding each as an (almost) black box. It maintains a large chunk of memory, the heap. On the heap, it allows to allocate and initialize appropriate object cells for any given class and implements, to reclaim unused memory, the garbage collector.

For object allocation, all existing implementations use a thread-local allocation buffer (TLAB), which reserves a small piece of memory for each thread to allow unsynchronized, blocking-free allocations. For the garbage collector, on the other hand, Maxine provides two different implementations: a mark-and-sweep GC and a semi-space heap that is explained in more detail here, as SqueakMaxine uses the semi-space heap and add new functionality.

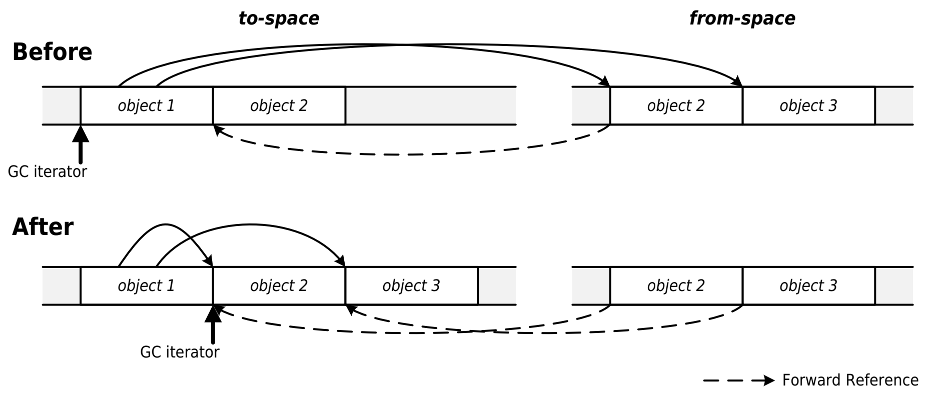


Figure 5.1.: GC step on the semi-space heap. Using forward references, the semi-space garbage collector updates all references in the object cell. If a referenced object was not reached yet, it is moved into the to-space.

On a semi-space heap, the memory is split into two regions: the from-space and the to-space. All active objects live in the to-space, this is also where the allocation of new objects takes place. Once the to-space is full, a GC run is triggered. The first thing that happens then is that the denotations of the spaces are swapped: the former from-space becomes the new to-space, and vice versa. Then, since all objects are now in the from-space, they have to be copied. This process is started by first copying all GC root objects: classes, threads and every object on the stack. For each object that was copied, the hub reference in the old cell is replaced with a forward reference to the new origin. This way, an object is never copied twice and the GC can find copied objects at their new address.

Once all root objects are moved, the garbage collector iterates over all objects in the to-space. For each object, all its references are updated, as illustrated in figure 5.1. To update a reference, the forward reference stored in the old cell is looked up and used as the new value, without the forward marking, of course. If an object is found that was not forwarded yet, it is copied to the end of the to-space. Once all references are processed, the GC moves on to the next object cell. This iteration is possible as the copying ensures that no unused spaces emerge between the objects. Eventually, every object that is reachable will be found this way.

It should be noted that the GC iterates from cell to cell, which means that to actually interpret the object's content, it has to rely on the layout to get the origin pointer for a cell.

5.1.3. Extended Extension Points

To avoid changing original sources as much as possible, the Squeak object model implementation was intended to be solely based on schemes . It hence seemed appropriate to use schemes as a configuration mechanism for Squeak-Maxine as well. However, a Squeak VM requires some features not present in a Java VM. Thus, some of the extension points, as they are defined by the schemes, were not sufficient. SqueakMaxine additionally requires a compiled method and a reference hybrid layout. To decouple these layouts from the rest of the VM an abstract class (`SqueakLayoutScheme`) was added. It can be used as a regular layout scheme by Maxine and, at the same time, ensures that SqueakMaxine can no longer be started with a Java layout implementation not extending this class.

In addition to the Squeak layout scheme there are two *de-facto* schemes that perform additional operations required for Squeak: the `SqueakRuntime` simplifies access to Squeak objects by defining a simple and optimized reflection APIand the `SqueakImageLoaderScheme` provides a method for loading objects from a Squeak image file into the heap. As adding new schemes to Maxine is not easily possible and, in this case, their implementations are quite depending on the layout, instances of these pseudo-schemes have to be obtained via the Squeak layout scheme.

The heap scheme also has to provide some additional methods. To support the image loading, the heap has to implement an additional interface, `ImageLoadingAllocator`, that allows to load a Squeak image file into memory and to reserve a large chunk of memory to allocate Squeak objects during the loading phase. Furthermore, the heap has to store the special objects array, which is a main GC root of Squeak. Just as with the layout scheme, a refined `SqueakHeapScheme` interface was introduced. A full list of all additional interface methods can be found in Appendix B.

5.2. The Squeak Layout Approach

The first approach to test the extensibility of Maxine was to re-implement the Squeak VM as close as possible, to see to which extend the existing parts of Maxine can be reused or adapted. For the object model, this means to support Squeak's cell layout and to use only existing meta-objects, as they can be loaded from the image, instead of Maxine's actors.

5.2.1. Implementing a Layout

Using Maxine's pointer API, implementing the Squeak layout is straight-forward. As an example, listing 5.3 shows how to convert cell and origin pointers. For the OHM layout, cell pointer and origin are identical, but the Squeak layout uses optional header words. Each header word uses its two lowest bits to indicate the header type. Thus, `cellToOrigin` just reads the first word and increases the cell pointer by the appropriate number of words.

```
blic Pointer cellToOrigin(Pointer cell) {
Word firstHeader = cell.getWord();
ObjectHeaderType headerType = ObjectHeaderType.detect(firstHeader);
return cell.plusWords(headerType.getExtraWords());


blic Pointer originToCell(Pointer origin) {
Word baseHeader = origin.getWord();
ObjectHeaderType headerType = ObjectHeaderType.detect(baseHeader);
return origin.minusWords(headerType.getExtraWords());
```

Listing 5.3: Origin to cell for the Squeak layout and vice versa.

Most layout methods, however, can have no meaningful implementation for the Squeak layout. This is caused by assumptions in the definition of the interface, such as that every layout implementation uses at least the header fields of the OHM layout: hub, misc and length; that there are exactly three layout kinds: tuple, hybrid, and array; or that the header fields of an object depend only on its layout. An overview over the implementation status of representative methods is given in table 5.1 for the general layout and in table 5.2 for the specific layouts.

Even though only a small part of the layout methods are implementable, implementing the Squeak object model is still possible. Most methods are used only during the bootstrapping phase, when support for Squeak objects is not required; by the Inspector and the Java reflection API, which are not supported for Squeak objects; and by the heap, which has to be adapted for Squeak anyway. The access to objects at runtime, on the other hand, is defined by code snippets which are compiled directly into machine code, without using layout methods as an intermediate step.

Thus, a full definition of the Squeak layout was realized in an independent class. Furthermore, the layout implementation is required in two variations:

Table 5.1.: `GeneralLayout` methods implemented for the Squeak layout.

Method	Description	Implemented	Comment
`cellToOrigin`, `originToCell`	Conversion between object and cell pointers.	✔	See example
`isTuple`, `isHybrid`, `isArray`, `category`	Identifies an object's Maxine layout kind.	✘	Used only by heap implementations, which make implicit assumptions on layout not compatible with Squeak.
`specificLayout`	Returns an object's specific layout implementation.	✔	The layout is determined from the object format encoded in the base header.
`size`	Returns an object's cell size in bytes.	✔	Reads either the size header or the size bits of the base header.
`readHubReference`, `writeHubReference`	Accesses the hub header field.	✘	Write access is required only for object allocation, which has to be rewritten anyway. Read access is required only for the inspector.
`readMisc`, `writeMisc`	Accesses the misc header field.	✘	Required for Java's synchronization feature; not supported for Squeak objects.
`forwarded`, `readForwardRef`, `writeForwardRef`	Handles forward references (used, i.e., by semi space garbage collector, to move objects).	✔	Sets an unused bit in the base header to flag a forward reference.

one version that provides fast and efficient access at runtime, and one version that performs validation to avoid loading corrupted image data.

This leads to the architecture shown in figure 5.2. The base class `ImageIO` defines the general layout rules, but does not specify how the image is accessed or how invalid data handled. As an example, listing 5.4 shows how `ImageIO` defines the check to determine whether an object has a compact class. The implementation is completed in the subclasses, as shown in listing 5.5 and listing 5.6, respectively. The `ImageAccess` class, one the one hand, implements efficient runtime access; here, the memory is accessed directly, based on the assumption that the data is correct and no exceptions are defined. For image loading, on the other hand, the second subclass `ImageLoader` always parses the entire object header to detect invalid references or inconsistent header data, and throws a checked exception if validation fails.

Table 5.2.: The `SpecificLayout` interface is too specialized for Java to allow an implementation of the Squeak layout.

Method	Description	Implemented	Comment
`isTupleLayout`, `isHybridLayout`, `isArrayLayout`, `category`	Identifies the Maxine layout kind represented by this layout.	✘	Used only by heap implementations, which make implicit assumptions on layout not compatible with Squeak.
`headerSize`, `headerFields`	Returns informations of the header fields used by this layout.	✘	In Squeak, the header fields are independent of the layout and depend on other properties of the object.
`visitObjectCell`, `readValue`, `writeValue`	Accesses the object on a low level.	✘	Required only by Inspector and during bootstrapping, not relevant for Squeak.
`layoutFields`, `getFieldOffsetInCe`	*Tuple and hybrid layout only.* Required for managing fields in a tuple.	✘	Requires field actor arguments, which are not used for Squeak.
`readLength`, `writeLength`, `arrayLengthOffset`	*Array and hybrid layout only.* Accesses the length header field.	✘	Array length has to be derived from cell size. Callers of these methods assume that there is a length header field.

5.2.2. Dual Semispace Heap

A problem that occurred during the implementation is that given an object reference, it is not possible to determine whether it references a Java or a Squeak object. Using `instanceof` is not possible, as we need this functionality to implement `instanceof` in the first place. Furthermore, without this distinction it is not possible for the garbage collector to access the object's fields. At the origin of a Java object is its hub header field, and at the origin of a Squeak object is its base header. Thus, it should be possible to check whether the word that is identified by the object reference references a hub. It is not likely that the base header bits of a Squeak object are set in a way that they form a valid hub reference. Nevertheless, it is technically possible, so this is not an adequate solution. In consequence, it is not possible to distinguish between a Java and a Squeak object with complete certainty just by inspecting its memory.

Another solution would be to derive this information from the context that references the object. It seems reasonable that Java objects and methods

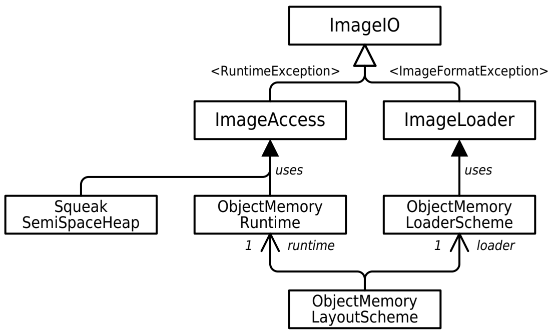


Figure 5.2.: Implementations of the Squeak layout. Two variations of the Squeak layout definitions are used, depending on the context.

```
public abstract class ImageIO<E extends Exception> {
 // 32 or 64 bit, little or big endian
 protected final ImageVersion version;

 public boolean hasCompactClass(SqueakReference obj) throws E {
  return getCompactClassID(obj) > 0;
 }

 protected abstract int getCompactClassID(SqueakReference obj)
  throws E;
}
```

Listing 5.4: `ImageIO` defines how values are found and interpreted, but not how they are accessed.

are allowed to reference only Java objects, and Squeak objects can reference only Squeak objects. However, this still does not solve the problems of the garbage collector. As described in section 5.1.2, the semi-space iterates over all objects and has to convert between cell and origin pointers, which is only possible if the object's general layout is known. Thus, it has to be ensured that Java and Squeak objects never mix with each other on the heap.

The easiest way to achieve this is to split the heap into a Java and a Squeak region (which are further each split into the to- from-space). This allows to run the GC individually for each region, using correct layout both times.

To implement this change, modifying the `SemiSpaceHeapScheme` of Maxine was necessary. By default, the existing implementation only supported collecting one region and accessed the layout via a global variable. Thus, region-specific details had to be extracted from the GC algorithm implementation.

```
public abstract class ImageAccess
  extends ImageIO<RuntimeException> {
  protected int getCompactClassID(SqueakReference obj) {
    final long baseHeader =version.readWord(obj, BASE_HEADER_OFFSET);
    return (baseHeader >> COMPACT_CLASS_SHIFT) & COMPACT_CLASS_MASK;
  }
}
```

Listing 5.5: In the `ImageAccess` implementation, the memory is accessed directly, using the Maxine Pointer API.

```
public abstract class ImageLoader
  extends ImageIO<ImageFormatException> {
  protected int getCompactClassID(SqueakReference obj)
    throws ImageFormatException {
    ObjectHeader header = ObjectHeader.read(obj, version); / may throw
    return header.getCompactClassId();
  }
}
```

Listing 5.6: The `ImageLoader` uses additional validation logic to check the integrity of the data and throws checked exceptions.

We introduced the garbage strategy, which hides the actual memory region, as well the logic how individual object cells are processed, from the algorithm. A generalization of this change is discussed in more detail as variation point "Garbage Strategy" in section 7.2.1. With the dedicated heap, loading an image file requires the same steps it does in Squeak. The file is directly copied into memory and, with a single iteration, all object references are updated to fit the new address range. The first few bytes of both from- and to-space are reserved for the image header and are not available for object allocation. To save the image, the image header bytes just have to be updated and then the entire active part of the to-space is written into a file. To minimize the file size, a GC run can be executed first.

5.2.3. Integration in Maxine

To trigger the image loading, the run scheme was replaced. Instead of invoking a main-method, it finds the image file and passes it to the image loader of the Squeak layout scheme. Then, the next step would be to begin executing the last active thread.

Hoewever, even without support for code execution, with a loaded image it is possible to print a stack trace of the thread that was executed last, as even the stack is stored as a chained list of Squeak objects. Shown in listing 5.7 is the algorithm to print such a stack trace.

```
 1 function printStackTrace()
 2   nil $\leftarrow$ SpecialObjects[NilObject]
 3   blockClass $\leftarrow$ SpecialObjects[ClassBlockContext]
 4   context $\leftarrow$ SpecialObjects[SchedulerAssociation]["value"]
 5         .activeProcess.suspendedContext
 6   while context $\neq$ nil do
 7     print("context:", context)
 8     print("receiver:", context.receiver)
 9     print("instruction pointer:", context.instructionPointer)
10     if context.getClass() = blockClass then
11       print("#args:", context["nargs"])
12       print("home:", context["home"])
13       print("startpc:", context["startpc"])
14     else
15       print("method:", context["method"])
16     end if
17     println()
18     context $\leftarrow$ context.sender
19   end while
20 end function
```

Listing 5.7: After the image is loaded, a stack trace of the last active thread can be printed (general algorithm).

First, the `nil` object and the closure class are fetched from the special objects array and cached for later use (line 2 and 3). In line 4, the last active execution context is retrieved from the scheduler. Now, it is possible to print information about the context (line 7–9), for instance the receiver (the object that can be accessed via `self`), or the current bytecode index. If the context is a closure, which is determined by checking the context's class in line 10, additional information is available (line 11–13): the number of arguments, the method that defines the closure, and the closure's first bytecode index within the bytecode array of the defining method. For regular method invocations, the invoked method can accessed to get additional information. Finally, the context's sender is fetched to move one level down the stack. The process is repeated until no previous sender is found.

For simplicity, the algorithm was shown in pseudo-code; the actual Java code is far less readable. As all operations are performed on Squeak objects, it is necessary to obtain the Squeak runtime first, which is done with

```
SqueakRuntime rt = SqueakConfiguration.runtime();
```

Then, it is possible to access a special object using

```
SqueakReference nil;
nil = rt.getSpecialObjectReference(SpecialObject.NilObject);
```

Accessing an instance variable can be done in two ways. For instance, it is possible to use a well known index that was defined in an enum before. Thus, `context.receiver` becomes

```
rt.getInstanceVariable(context, ContextFieldIndex.RECEIVER)
```

A more flexible way to access a field is to specify its name. In Java, pseudo-code similar to `context["home"]` is implemented as

```
rt.getInstanceVariable(context, "home")
```

To access an object's class, `rt.getClass(context)` can be used. Finally, to print an object, it has to be converted into a Java `String` object first by using `rt.toJavaString(context)`.

A special challenge of a Squeak implementation are two methods, which are required by some meta-programs and VM-internals. The first method, `firstobject`, returns the first object, for an arbirary definition of first, while `nextobject` returns the successor of a given object, again for an unspecified ordering. Together, these methods can be used to iterate over all objects on the heap, although objects may be skipped or found multiple times, for instance, if a GC run takes places.

With the dual heap, both methods are easy to implement. The first object is the object located at the beginning of the to-space, while the next object can be found by going to the end of the current object's cell. The semi-space GC ensures that this will always work, as no empty spaces between objects can occur.

For easier debugging of SqueakMaxine, it would be desirable to browse Squeak objects in the Inspector. However, the Inspector requires class actors and has strong assumptions about the layout implementation. Thus, while it is possible to view the memory of the Squeak object space, objects have to be interpreted manually.

Adding or removing fields in a class is pending implementation, because it requires updating all instances of this class, including subclasses. With the new garbage strategy, this can be achieved easily. To update all objects, a special GC run is triggered, which uses a different garbage strategy to change the way how objects are processed. For instances of the modified class, instead of simply copying the cell from one space to the other, a word is inserted or removed to realize the field modification.

Even though this approach worked well as an object model implementation, severe problems started to appear with outlook to the integration of the execution model. Due to the way bytecodes are compiled from Java code-snippets, it was no longer possible to keep the distinction between the two object worlds. In parts, this was also caused by a lack of separation between compiler and layout, which would have required re-implementing large parts of the layout. All in all, this approach seemed to require too much work to be worth continuing, although we believe it should be possible to implement a fully working VM this way. Relevant for this work is that another object model implementation was needed, which had to provide more compatibility with Maxine's existing understanding of objects.

5.3. The Maxine Layout Approach

The second approach implements the object layout presented in section 3.3.3. It creates class actors for all Squeak classes and uses, when possible, the same layouts for Squeak objects that Maxine uses for Java objects.

5.3.1. Extending the layout

Copying and adapting the OHM to become the YOHM layout was a rather simple task. The first change was to add another header field to the tuple layout. As listing 5.8 shows, only minimal changes were necessary for this: a new header field object is created (line 1), which handily provides meta information for the inspector; the header offset is adjusted to include an additional word (lines 3–9); and the new field is included in two methods that describe the header layout to other parts of the VM (lines 11–21).

The second change added two new layout kinds: the hybrid reference array layout and the compiled method layout. These layouts differ in only one way from existing layouts (hybrid with integer array, and byte array): their

```java
public static final HeaderField CLASS_ACTOR = new HeaderField(
  "classActor",
  "to align tuples and store class actors of class objects");

private final int headerSize = 3 * Word.size();

public final int classActorOffset;

public YHomTupleLayout() {
  classActorOffset = miscOffset + Word.size();
}

public HeaderField[] headerFields() {
  return new HeaderField[]
    {HeaderField.HUB, HeaderField.MISC, CLASS_ACTOR};
}

@Override
public Offset getOffsetFromOrigin(HeaderField headerField) {
  if (headerField == CLASS_ACTOR) {
    return Offset.fromInt(classActorOffset);
  }
  return super.getOffsetFromOrigin(headerField);
}
```

Listing 5.8: Minimal changes were necessary to create the YOHM tuple layout from the existing OHM implementation.

semantic implies object references in the object cell that are not known to the object's class (cf. variation point "Reference Visitor"). These layouts are instantiated by the layout scheme and ignored by the Java part of the VM.

However, to actually use the YOHM layout for Squeak objects, the Squeak class actor is required. On the one hand, it has to act as a bridge from a Squeak object to its Squeak class, on the other hand, it has to represent the Squeak class in terms that Maxine can understand. Maxine has different class actor implementations for different types of classes. The superclass for all actors that represent instantiatable classes, as opposed to primitive types and interfaces, is `ReferenceClassActor`. It is further specialized into subclasses for each layout kind. To reuse the object-kind specific class actor code, Squeak class actors were added as further subclasses. As figure 5.3 shows, the generic, layout agnostic Squeak class actor then had to be defined as an interface, which lead to a small amount of code duplication for handling references to the Squeak class object.

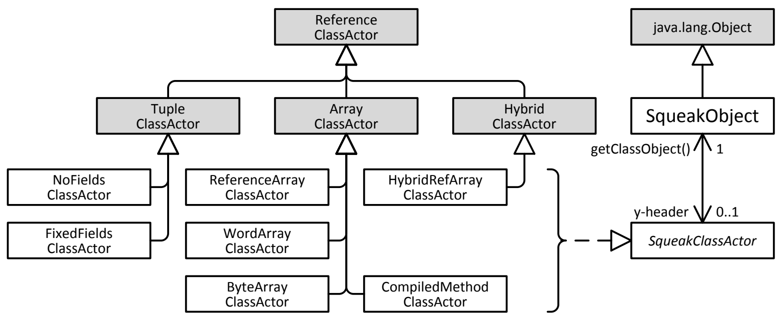


Figure 5.3.: The Squeak class actor hierarchy. Original classes are highlighted.

The initialization of a class actor requires a full class description. The layout kind is already chosen by instantiating the appropriate Class actor class, but class name, super class, field names for fixed field and hybrid classes, and method actors have to be passed as parameters.

The *class name* has to be provided as a Java string, which can be created by locating and converting the bytes of the Squeak string specifying the class name. The *super class* is identified by its class actor. In consequence, the class load order is determined by the inheritance tree; root classes have to be loaded first. The representation of *fields* for Squeak is even simpler than it is for Java. In both Squeak and Java, a field has a name and identifies some bytes after the objects origin. In Java, however, fields may have different widths (one to eight bytes) which requires the field offset to regard the widths of previous fields. In Squeak, all fields store references and have the same width: one word (eight bytes in 64-bit). Thus, to set up a Squeak class actor, an array of field actors can be directly created from the array of field names; the fields' offsets and types are not needed to be explicitly specified, and there is no need for a dedicated Squeak field actor.

For *methods*, the signature definition is of similar simplicity: all that is required to describe a method is its name and the number of parameters. Squeak allows characters in method names that are not allowed in Java, therefore, a bijective name mapping has to be applied. The current implementation simply replaces non-letter characters with a dollar sign, followed by the character's ASCII code; `ifTrue:` becomes `ifTrue$36`, for instance.

With this setup, it is again possible to handle Squeak objects with Maxine, which includes accessing fields and invoking methods. However, this object model implementation is still useless without the capability to load objects from an image file.

5.3.2. Image Loading

Loading objects from a Squeak image is much more complex with this layout approach. Unlike as with the Squeak layout, where only an offset had to be added to all references to have them point into the new memory region, this time the objects are converted into the YOHM layout, which also includes the creation of class, field and method actors. From previously one to three, now every object has three header words. Furthermore, when loading a 32-bit image, all fields have to be converted to 64 bit. This does not affect byte, word and long arrays, but doubles the content size for fixed field, hybrid and reference array objects, and arbitrarily affects compiled methods, depending on their number of literal references.

As a result, without deeply examining the Squeak image first, neither can be predicted at which address a Squeak object will end up after conversion, nor is it possible to tell the total required amount of memory to hold the converted image. The former means that the loading process has to populate the heap with object cells which cannot be immediately initialized and thus are not processable by the GC. The latter is a problem because increasing the heap size requires a garbage collector run. In consequence, a sufficient amount of memory has to be allocated before starting the image conversion.

Fortunately, an upper bound of the possible required heap space can be derived from the original image size times two, plus two words for every object. This of course requires that the conversion algorithm creates only little garbage, which should not be taken for granted for a Java algorithm, and that actors are created when garbage collection and heap growing is possible. The latter is necessary anyway, as initializing class actors for about 4000 classes of a regular Squeak image creates over 400 MB of garbage objects.

This results in a five-stage loading procedure, as summarized in table 5.3. The first four steps each require an iteration over all objects in the Squeak image, and only in the first and the last stage garbage collection is possible. The remainder of this subsection will describe the five stages in more detail.

Again, the loading procedure is wrapped in a pseudo-scheme implementation, in this case in `squeakmaxine.image.max.MaxImageLoaderScheme`. Given the image file, a VM operation is started to ensure that all other threads are paused. Within the operation, the loading allocator of the Squeak heap scheme is used to map the file into memory. Then the first stage is entered.

Stage 1: Create class, field and method actors

To create objects in Maxine, a class, represented by a class actor, is required. As mentioned before, creating and initializing class actors creates a large amount of garbage objects and thus has to happen before garbage collection is disabled. In consequence, the first stage has to identify all behavior objects in the image, including classes, meta classes, traits and meta traits. To do this, the `Behavior` class has to be identified first, for it is a common super class of these, and only these, four types of objects. To find `Behavior`, the image data has to be interpreted, using the algorithm in listing 5.9.

Directly after opening the image file, the special objects array is the only Squeak object known by the VM, as it is referenced in the image header. In the special objects array, other objects, such as the `Integer` class, can be found at well defined indices. While `Integer` itself does not extend `Behavior`, its class does (similar to `Point` and `Point class` in figure 3.2). Thus, starting with the integer meta class, super classes are fetched until one with the name "Behavior" is found. Having identified `Behavior`, it is now possible to find all behavior objects by iterating over the entire image and checking for each object whether its class extends `Behavior`. If such an object is found, a class actor representation needs to be created. Since, to create a class actor, the super class actor is needed, a *class-name to class-actor* map is used to avoid creating a class actor twice. This is especially important as it cannot be expected that the class tree is stored in the image in an ordered fashion.

When a class actor does not yet exist, the behavior's method dictionary is read to collect Squeak method actors, each initialized with the method name and parameter count. For fixed field and hybrid classes, the field names are collected as well. Finally, a Squeak class actor is created and registered in the class actor map. Once all objects in the image have been looked at, it is guaranteed that all class actors exist and the second stage can commence.

Table 5.3.: The image loading process is divided into five phases.

Stage	Description	Full image iteration	GC enabled
1	Create all actors	✔	✔
2	Allocate object cells	✔	✘
3	Link Squeak meta objects to actors	✔	✘
4	Convert objects	✔	✘
5	Initialize method actors	✘	✔

```
behaviorNameBytes ← toByteArray('Behavior')
classNameBytes ← new byte[behaviorNameBytes.length]

function findBehaviorClass()
  aClass ← SpecialObjects[ClassInteger]
  aMetaClass ← aClass.getClass()
  while aMetaClass ≠ nil do
    if nameIsBehavior(aMetaClass) then
      return aMetaClass
    end if
    aMetaClass ← aMetaClass.getSuperClass()
  end while
  error 'Behavior not found'
end function

function nameIsBehavior(aClass)
  className ← aClass.className
  className.getBytes(classNameBytes)
  // fills classNameBytes with content of string
  return behaviorNameBytes = classNameBytes
end function
```

Listing 5.9: Finding the `Behavior` class by interpreting the image. Same pseudo-code as listing 5.7.

Stage 2: Allocate a cell for each object

In the second stage, object cells are allocated in the to-space. The cell size required by the YOHM layout can be easily derived from the Squeak object, as will be shown at the example of a compiled method in figure 5.4. Independently of the Squeak header size, three header words are required for every object in Maxine. Furthermore, the content words of fixed fields, hybrids, reference and word arrays, and compiled methods have to be adjusted to the new word width. Finally, the bytes of byte arrays and compiled methods do not change, although the number of alignment bytes might increase.

As mentioned before, while it is possible the calculate the object cell's size, its address in the to-space cannot be predicted. Instead, some association between the original Squeak cell and the target cell has to be established. Maintaining a hash map to map the hundreds of thousands of objects in a simple Squeak image would be inefficient and require an unpredictable amount of memory, which is not acceptable while garbage collection is disabled.

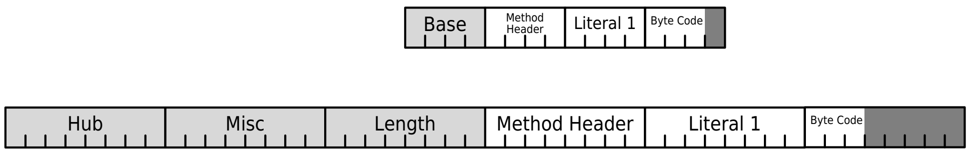


Figure 5.4.: When a 32-bit Squeak cell is converted into the 64-bit YOHM layout, it is impossible to predict the new size without analyzing the object, as is shown here for a compiled method with one literal and three bytecode bytes.

As an alternative, a reference to the target cell can be stored in the Squeak cell itself. Since the smallest possible Squeak object, with a compact class and no fields, would consist only of a base header, the base header field has to be used to store this reference. However, the base header must not be overwritten as it is still needed for subsequent image iterations and interpretations. The solution to this problem is illustrated in figure 5.5. The upper part of the figure shows a small part of an exemplary Squeak image, with an object whose cell was just allocated in the to-space, followed by a word array it references and its class. The base header was replaced with a reference into the to-space, shown in the lower part of the figure. By copying the base header value into the to-space, interpreting the image is still possible; the base header field just has to be dereferenced to obtain the actual value.

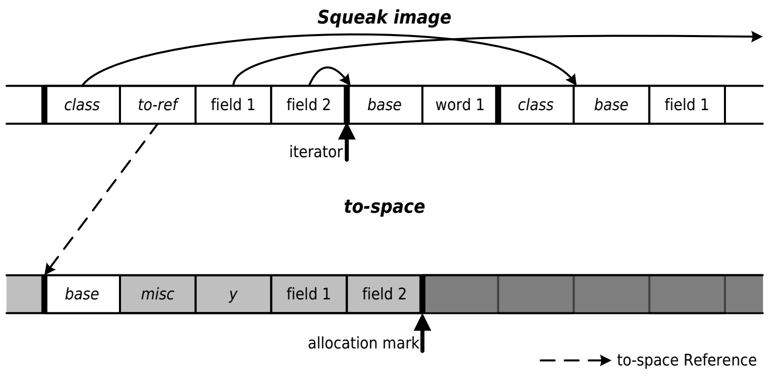


Figure 5.5.: An object cell is allocated for a Squeak object, referening its class and two other objects. A reference to the new cell is stored in the base header field. Uninitialized memory is shown in light gray, unallocated memory in dark gray.

Stage 3: Link Squeak meta objects to their respective actors

After the second step, the to-space contains data that is relevant for interpreting the Squeak image. Thus, initializing the target cells will render the image data unreadable. However, to set up an object its class's class actor is required, which can only be obtained via the original Squeak class object.

Therefore, just as in the first stage, all behavior objects are searched, but this time to link each class's target cell with the Squeak class actor. This includes storing a reference to the Squeak class actor in the additional header field in the target cell and invoking `setClassObject` on the class actor. Notice that the method expects a `SqueakObject` instance, whereas the cell does not contain valid object data, which would require at least a hub. Nevertheless, it still possible to pass the uninitialized object as a parameter, using Maxine's unsafe-cast feature to circumvent Java's type-safety checks. In a similar way, a unidirectional link is established from Squeak method actors to the future compiled method objects.

Stage 4: Convert objects into YOHM layout

With the class actors in place it is finally possible to convert all objects from Squeak into the YOHM layout. To pick up the example of figure 5.5, the same object is shown in figure 5.6 after its conversion.

For each object, its class's target cell is used to look up the class actor and fill the hub header field. For variable sized objects, the length is calculated from the original Squeak cell size. Finally, all references to other objects are resolved to point to their respective cells in the to-space, while word and byte data is copied.

Stage 5: Initialize method actors

Now, as all Squeak objects have been properly set up in the to-space, the image data can be discarded and garbage collection is possible again. However, while the objects and classes are functional already, the Squeak method actors require further initialization. Thus, iterating over the class actors collected in stage 1, each of its Squeak method actors is triggered to collect information that is required by the compiler, e.g., the number of arguments and the byte code array. With this step completed, the image is fully loaded and the objects and classes can be handled by the VM.

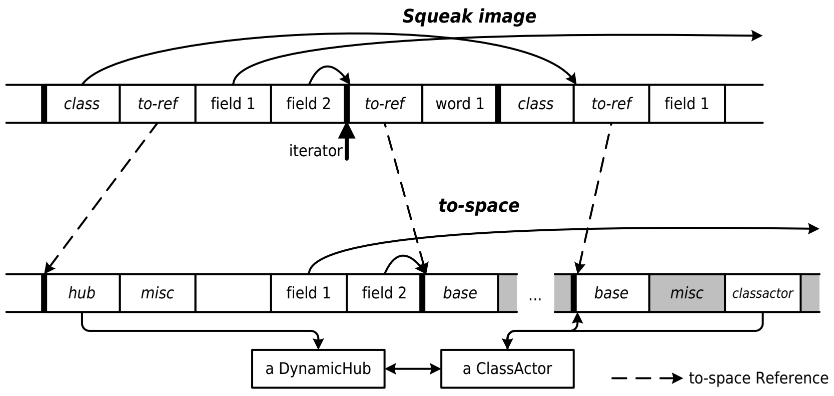


Figure 5.6.: After a Squeak object converted into the YOHM layout, all its references point into the to-space, even if the targeted objects are not initialized yet. The dynamic hub is obtained via the class object.

5.3.3. Integration in Maxine

With a working image loading process, the object model is ready for execution. It is also possible to operate manually on Squeak objects, using the Java reflection API. This way, it is easy to implement the Squeak runtime interface, which allows printing a stack trace using the same code as for the Squeak layout (cf. listing 5.7).

To debug the code and the subsequent execution model implementation, it was necessary to be able to examine Squeak objects in the Inspector. Since the Inspector uses the same layout implementation as the VM itself does, not much work was expected here. To correctly display an object's contents, however, the Inspector also needs to know details about fields and methods of the object's class , which it does not obtain from the actors present in the VM, but from reading the class file, which does not exist for Squeak objects. Fortunately, the functionality to create class files for runtime-generated classes already exists in Maxine. Furthermore, the object models of Squeak and Java are similar enough to allow for Squeak classes being described in class files. Thus, when the image is loaded, a class file describing superclass, fields and method signatures (bytecodes are not included) is generated for each Squeak class. As described beforehand, some parts of Squeak require the two functions `firstobject` and `nextobject`. Again, the semi-space GC ensures that the objects on the heap can iterated easily. With this approach, however, Java objects have to be skipped explicitly.

As with the Squeak layout approach, the modification of classes is not supported. Even though the Maxine actors should mirror the Squeak meta objects, they are not updated when the latter are changed. Due to the actor system's complexity, it is not sufficient to simply remove the final-modifiers from the actor's fields. Alas, a comprehensive analysis of necessary preconditions and side effects of altering classes and recompiling methods was not possible in the time of this project.

5.4. Tagged Integers

As described in section 3.4, tagged integers are special references that directly encode a value instead of pointing to a valid memory address. They are essential for an efficient implementation of numbers in a system where *everything is an object*.

A tagged integer is a reference, in the sense that it is a value that identifies an object, in this case an instance of `SmallInteger`. Thus, their characteristics are defined in the reference scheme, which was extended for this purpose (cf. table B.1). The method `isTaggedInteger` identifies such references by checking the smallest bit, `toInt` and `toLong` implement conversion into the respective primitive types of Java, and `fromLong` can be used to create a tagged reference for a given value. Even though small integers are 31-bit values, this forced conversion to `long` is necessary to ensure the sign of negative numbers is not lost when converting to 64-bit words. Thus, theoretically, SqueakMaxine supports 63-bit tagged integer values, but it was not extensively tested whether this breaks any assumptions in existing Squeak code.

Support in the Layout The integration of these methods into the system was different for both layout approaches. For the Squeak layout approach, tagged integers were explicitly implemented in the Squeak runtime, which was used to print the stack trace. Both methods `getClass` and `toJavaString` begin with checking the reference for tagging and handle such values differently: the former returns the small-integer class from the special objects array, while the latter creates a string from the object's primitive value.

For the YOHM layout implementation, tagged integers have to be regarded wherever header fields are accessed without checking the object's type first. Obviously, this happens in the header-field accessors of the general layout. To get the hub of a tagged reference, instead of trying to read the hub field, the small-integer hub is returned directly. For this purpose, the hub is stored in a private field of the YOHM general layout, which is initialized directly after loading the Squeak image. The hub field's setter is not affected, as it is called only during object allocation, which never happens for small integers.

The misc field accessors are not changed as well, as it is not possible to store additional data in a tagged reference. This has two consequences. First, it is not possible to synchronize on small integers. This should be no problem as this kind of synchronization does not exist in Squeak anyway. If Squeak objects are used in Java code, the programmer has to be aware to only synchronize on objects that known not to be small integers. Again, this should be no problem, as it is generally not advisable to synchronize on objects of unknown origin. Second, it is not possible for tagged integers to have an identity hash-code. To circumvent this problem, `SqueakObject`'s `hashCode` method was overwritten to return the integer value instead (cf. listing 5.10). The method `toString` was adapted in a similar way to return the integers value as a decimal string, instead of its class and hash-code.

The hub access definition in the compiler has to be changed as well; then, it is possible to use tagged integers just like real objects, as listing 5.11 shows.

```java
public final int hashCode() {
  if (SqueakReference.isTaggedInteger(Reference.fromJava(this))) {
    // I am a tagged integer, return plain value
    return Reference.fromJava(this).toOrigin().toInt() >> 1;
  }
  return super.hashCode();
}
```

Listing 5.10: `SqueakObject.hashCode()`. The method checks whether it is invoked for a tagged integer.

```java
Object value = Reference.fromOrigin(Pointer.fromInt(0x1337)).toJava();
value.getClass(); // returns SmallInteger class
value.toString(); // returns "2459"
System.out.println(value); // prints 2459
```

Listing 5.11: Tagged references can be used in regular Java code. The variable `value` literally contains the value `0x1337`, which is not a valid pointer, but the tagged representation of the number 2459.

Support in the Heap Other layout methods that directly access an object's hub header field are related to forwarding, when an object is moved by the GC. There is no meaningful implementation to these methods. Tagged integers cannot be forwarded and it should not be the layout's responsibility to pretend that tagged integers can be moved like regular objects.

Thus, reference handling in the heap had to be changed. Whenever it is checked for a reference if the object was already forwarded, it is first checked whether the reference is tagged. If this is the case, no forwarding happens. Thus, the semi-space GC never attempts to move a small integer object from one space to another.

It should be noted that a small integer value (especially for multiples of four) looks just like a forward reference. However, this ambiguity creates no problems as forward references can occur only in the hub header field, whereas small integers can occur everywhere *except* for header fields.

⁂

As has been shown in three instances, implementing a new concept *could* have been possible by replacing only one of Maxine's subsystems, but required to add specialized behavior in otherwise unrelated subsystems as well. This is caused by general assumption about how subsystems are implemented, and which features they provide or support.

These assumptions, of course, are reasonable for a single language VM, but greatly decrease the usefulness of a VM framework. We believe, however, that many of them can be removed with moderate to little effort.

6. Execution in SqueakMaxine

This chapter is to give insight into the application of the concept introduced in chapter 4 to the Squeak / Smalltalk System [21], a free and open-source dynamic object-oriented language derived from the original Smalltalk-80 [14]. It provides the challenges mentioned in section 2.2 and, thus, is suitable to describe the implementation details of applying the ideas from section 4.1 and to evaluate them. The resulting VM is called *SqueakMaxine*.

This chapter is organized as follows: for every aspect of execution model implementation and the VMs architecture, we give the Maxine way of implementing the aspect for Java and subsequently, the approach taken for SqueakMaxine; with the required additions, adaptions, and modifications.

6.1. Architecture of SqueakMaxine

The major components of Maxine are shown in figure 6.1, all of which will be explained throughout this chapter. While Maxine has no deliberate layering approach, certain layers in Maxine exist nonetheless. Being interested in the execution model implementation, we treat the object model implementation as a black box whose outlets are utility classes like `Layout`, their configuration schemes like `LayoutScheme`, and the Maxine internal representations of language entities, the *actors*. These elements form the ground layer on which the execution model implementation builds. Additionally, the system provided JDK, accessed trough the Java native interface (JNI), forms the source built-in behavior and, hence, is an integral part for Maxine. On top of this foundational layer, the *runtimes* provide functionality that is to be accessed by native code compiled methods, but not directly compiled into them. Examples include allocation or exception handling. On top of these, the JITs operate, generating native code for methods passed from the guest language. A broker coordinates, which method is compiled when by which compiler and is eventually called from the run scheme that defines the basic start-up and initial guest language code invocation.

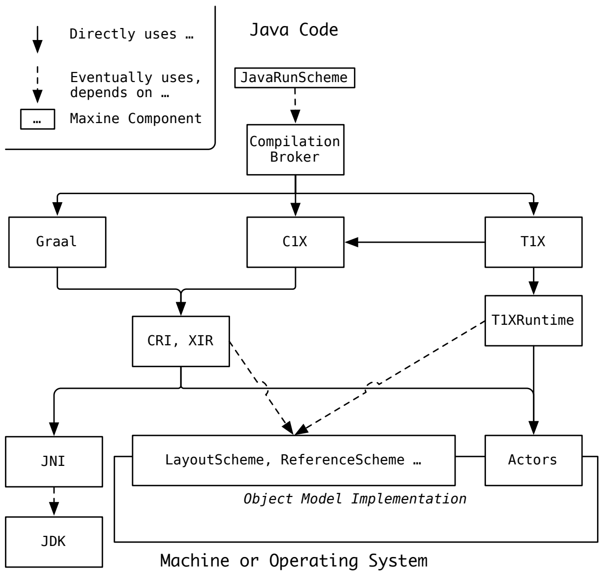

Directly uses …
Eventually uses, depends on …
Maxine Component
Java Code
JavaRunScheme
Compilation Broker
Graal
C1X
T1X
CRI, XIR
T1XRuntime
JNI
JDK
LayoutScheme, ReferenceScheme …
Actors
Object Model Implementation
Machine or Operating System

Figure 6.1.: Architecture of the Maxine Execution Model Implementation. Auxiliaries like assemblers are omitted for brevity. Key: as indicated; "Maxine Component" refers to semantic units of object providing the indicated functionality, e.g., "Actors" is all instances of subclasses of `Actor`, such as `MethodActor`.

6.1.1. Changes by SqueakMaxine

Pursuing the goal of using Maxine as a framework for dynamic object-oriented language VMs, there are only minimal changes to the Maxine architecture itself. As illustrated in figure 6.2, Maxine is merely augmented by the Squeak specific entities required. Quite obvious is the parallelism between the part under `T1X` and under `S1X`. Both being JITs for Java and Squeak respectively, they are related, as to be explained in section 6.3. Other additions include a new compilation broker, additional required actors and a run scheme that interacts with the object model implementation to load Squeak image files and invoke behavior from them.

6.2. Runtime Object Access

Providing a runtime interface for object access is subject to the object model implementation chosen. Thus, Maxine provides the following facilities for object access:

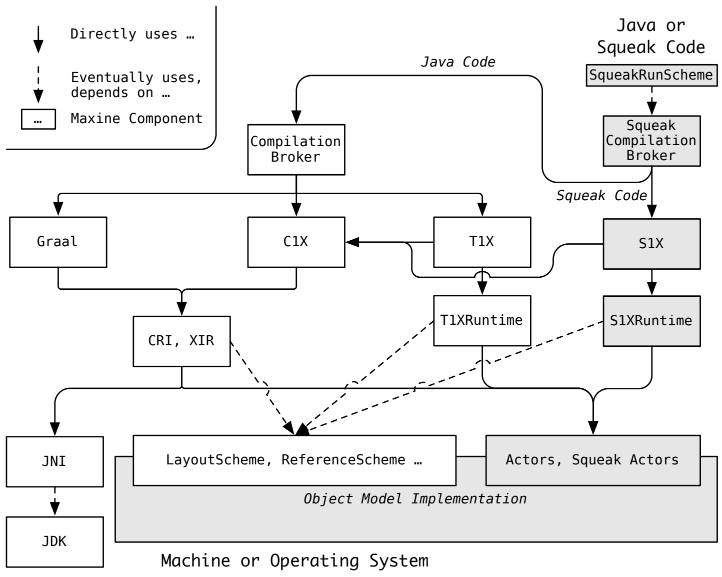


Figure 6.2.: Architecture of the SqueakMaxine Execution Model Implementation. Key: as in figure 6.1; additionally, grey shading indicated addition or modification for SqueakMaxine.

Heap access The `Heap` class provides methods for object creation, i.e., allocation.

Accessors `Accessors`, i.e., `Pointers` and `References`, provide direct memory alteration. To achieve this, they are treated specially by Maxine's compilers.

Layouts The `Layout` utility class abstracts from the chosen object layout and dispatches to the concrete layout (e.g., of an array or tuple object), which, in turn, uses the `Accessor` API.

Accessory classes The accessory classes `TupleAccess` and `ArrayAccess` provide facades for Java object to the `Layout` class that operates on `References` only. The `ObjectAccess` class provides general acces to objects, including reading objects sizes or certain header fields. By the layout nature of Maxine objects, the latter includes resolving the class of an object.

Snippets The more general `Snippets` utility class includes, among other functionality, methods for resolving classes and creating objects. To achieve this, it uses `ResolvingGuards` and some of the accessory classes for resolving and the `Heap` class for creation.

6.2.1. Maxine

Runtime object access is necessary primarily for Java bytecodes that modify objects, hence, these facilities are normally only accessed through compiled code. The JITs, therefore, compile invocation of behavior of the mentioned classes into compiled methods. For baseline compilation, templates are used that are compiled by the optimizing compiler and implement object modifying bytecodes through ACCESSORY CLASSES. Additionally, the templates for resolving classes and object creation use SNIPPETS and the `ObjectAccess` accessory class. However, due to the inlining the optimizing compiler does, all these calls are compiled to direct native machine code instructions altering objects. This whole process is similar for the optimizing compiler with the notable difference that no templates exist, thus, the object access is specified in special compiler and runtime independent intermediate representation (XIR) snippets, written in a higher level assembler, generating the memory alteration instructions directly. Note that this circumstance currently breaks the abstraction intended by the LAYOUT abstraction.

6.2.2. SqueakMaxine and the `YOhmLayout`

Likewise to Maxine, runtime object access in SqueakMaxine is used for implementing the respective Squeak bytecodes. Hence, the template source for Squeak uses the main facility of SqueakMaxine for object access: the `SqueakRuntime`. Similar to the accessory classes described previously, this class encapsulates all calls to the `Layout` class.

The `SqueakRuntime` and `Layout` abstractions are not completely suitable for implementing *class object* access. Class objects are the Squeak domain objects representing classes. In the object model implementation, there is a dualism between Squeak class objects and Maxine `ClassActor` instances and it is necessary to refer from one representation to the other. While the path `ClassActor` instance to class object is easy to provide via an extra field in the already present Squeak specific subclasses of `ClassActor`, the path vice versa is harder to achieve. The reference to the class actor is stored in a header field of the Squeak object, whose layout is defined by the `YOhmLayout`. That header field, however is not accessible using means like `Layout`. In the cases where accessing the class actor from the class object is necessary, the `SqueakRuntime` and `SqueakTemplateSource` therefore circumvent the `Layout` abstraction and use the functionality of the `YOhmLayout` directly.

6.3. Bytecode Compilation

The most significant part of the execution model implementation in Maxine is the *bytecode compilers* which transform a given language-specific intermediate representation (i.e., bytecode) into machine code. As this is always done on the fly, all of Maxine's bytecode compilers are just-in-time compilers (JITs) and as such actually provide an IR implementation in the sense of section 4.3.

The Need for Compiler Selection Maxine naturally supports more than one JIT. Since, as a design decision, no Java bytecode interpreter is included in Maxine but Java reflection and debugging facilities need information like the current bytecode index, a compiler is necessary that can produce code with this information. Such a compiler in Maxine is called *baseline compiler*, or interpreter-compatible. Furthermore, Maxine has a notion of optimizing *optimizing compilers*, which not necessarily need to provide as excessive runtime information as a baseline compiler. The coexistence of two different compilers calls for a management facility that can determine which compiler to use for compiling a certain method. The `CompilationBroker` is just this facility in Maxine. The broker selects the initial compiler for a method and the compiler used for re-compilation of an already compiled method; and may trigger fail-over compilation, i.e., methods that fail to compile with one compiler are again compiled with another one. The `RuntimeCompiler` interface along with a `CompilationBroker` pose the entry point to the multi-JIT infrastructure of Maxine.

6.3.1. Java Compilation with `C1X` and `T1X`

Maxine provides no interpeter for Java but two JITs, one optimizing compiler (`C1X`) and one baseline compiler (`T1X`). Also, the default `CompilationBroker` is part of the Java compilation facility. This broker is setup in a way that all methods to be compiled are compiled first with the baseline compiler and — under certain circumstances like enough invocation — later-on with the optimizing compiler. Furthermore, the respective other compiler is set up as fail-over compiler. In any case, fail-over recompilation is only tired once (cf. figure 6.3).

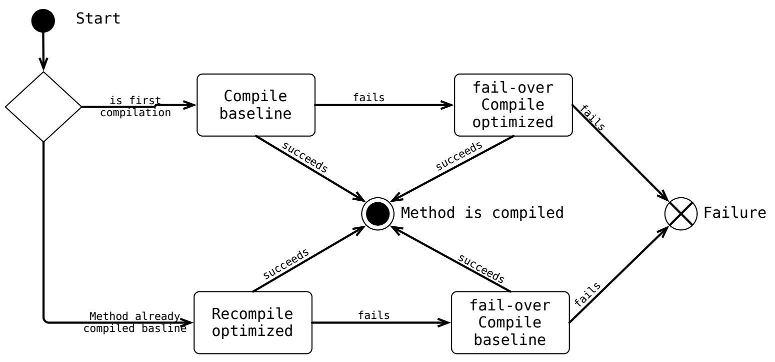


Figure 6.3.: Default Java compilation broker behavior. On first pass a method will be compiled using the baseline compiler, else the optimizing compiler. A fail-over step is done in case the first compilation fails. Key: UML state diagram.

The Optimizing Compiler, `C1X` For optimized compilation, Maxine uses `C1X`, a JIT derived from the HotSpot™ client compiler named *C1* [23, 31]. It was ported from C++ to Java, and improved [43] as well as extended with a compiler–runtime interface (CRI) and an intermediate representation, the *compiler and runtime independent intermediate representation* (XIR) [48, 40]. The higher-level assembly part of `C1X` and much of the built-in behavior definitions are implemented by means of XIR.

`C1X` is not required to compile every possible execution path in a method. The compiler with rather compile the *hot path* of a method with aggressive optimizations — especially inlining — and will *de-optimize* [19] dynamically for more uncommon paths [32, 23, section 2.6]. Unlike the standard JVM that reverts to interpreted execution, Maxine in this case uses a baseline compiled version of the method.

The `C1X` compiler implements extensions to the Java language as explained in section 2.1.1; `Word` data types are mapped to corresponding low level data types. It is hence the bootstrap compiler of Maxine and therefore used for building boot images for a Maxine VM. This also applies to the compiler; `C1X` is compiled by itself during boot image generation.

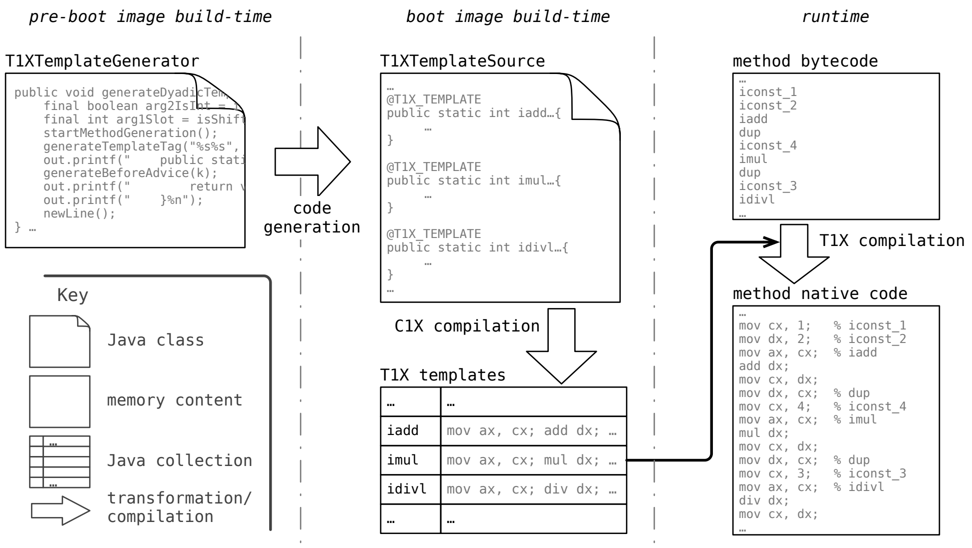
We did not deeply investigate `C1X` with respect to variation points, as its implemented optimizations are very Java-specific and time-intensive to bring to other languages. To show that implementing execution models of dynamic object-oriented languages is possible with Maxine, an optimizing compiler is not necessarily required. Hence, we proceed with the non-optimizing compiler, `T1X`.

The Baseline Compiler, T1X Ordinarily, a Maxine VM does not contain an Java interpreter as, e.g., the standard JVM does. However, there are situations where an optimized compilation as done by C1X does not fit. This includes the initial compilation of a method (costly to be done with C1X for every method) and de-optimization (see the previous section), among others. To support this non-optimized execution of methods in the absence of an interpreter, Maxine has a notion of *baseline* compilers that produce non-optimized code that is sticking to the bytecode semantics strictly. For Java, this behavior is also called *interpreter compatible*.

The Java baseline compiler in Maxine is a template-based bytecode compiler named T1X, a portmanteau of "template-based" and "using C1X". Template-based compilers produce native code through collating pre-built machine code snippets, the templates, often using one template per instruction. The T1X compiler is such a template compiler. It has templates for more than half of the Java bytecodes. However, instead of using handwritten templates for that bytecodes, T1X rather has Java written templates. These are compiled by the C1X compiler, hence the "1X" in the name. Refer to figure 6.4 for the transformation process from template source code to native code templates to native compiled methods.

```
/* T1XTemplateSource part for IADD */

@T1X_TEMPLATE(IADD)
public static int iadd(@Slot(1) int value1, @Slot(0) int value2) {
    return value1 + value2;
}

// --- --- ---

/* T1XCompilation part for IADD */

protected void processBytecode(int opcode) throws InternalError {
    beginBytecode(opcode);
    switch (opcode) {
        // …
        case Bytecodes.IADD    : do_iadd(); break;
        // …
    }
    // …
}

protected void do_iadd() {
    emit(IADD);
}
```

Listing 6.1: T1X parts relevant for the IADD bytecode.



```
public void generateDyadicTem
    final boolean arg2IsInt = 
    final int arg1Slot = isShift
    startMethodGeneration();
    generateTemplateTag("%s%s",
    out.printf("    public stati
    generateBeforeAdvice(k);
    out.printf("        return v
    out.printf("    }%n");
    newLine();
} …
```

```
…
@T1X_TEMPLATE
public static int iadd…{
     …
}

@T1X_TEMPLATE
public static int imul…{
     …
}

@T1X_TEMPLATE
public static int idivl…{
     …
}
…
```

…	…
iadd	mov ax, cx; add dx; …
imul	mov ax, cx; mul dx; …
idivl	mov ax, cx; div dx; …
…	…

```
…
iconst_1
iconst_2
iadd
dup
iconst_4
imul
dup
iconst_3
idivl
…
```

```
…
mov cx, 1;   % iconst_1
mov dx, 2;   % iconst_2
mov ax, cx;  % iadd
add dx;
mov cx, dx;
mov dx, cx;  % dup
mov cx, 4;   % iconst_4
mov ax, cx;  % imul
mul dx;
mov cx, dx;
mov dx, cx;  % dup
mov cx, 3;   % iconst_3
mov ax, cx;  % idivl
div dx;
mov cx, dx;
…
```

Figure 6.4.: `T1X` templates overview. `T1X` templates are Java code snippets that are compiled to native code by the `C1X` compiler (middle) and then used during `T1X` compilation of Java methods (right). Several Java bytecodes exist in variants, e.g., multiplication for different types; the template source for those bytecodes is generated prior to template compilation (left). Key: as indicated.

As mentioned, the templates for the `T1X` compiler initially are Java code snippets that are compiled to native code. `T1X` uses `T1XTemplateSource`, a utility class where all theses templates are gathered. A template in the `T1X` sense now is a **`static`** method that has a `@T1X_TEMPLATE` annotation. Find an example of a template in listing 6.1. The argument to the annotation specifies the implemented bytecode as a tag, which however does not need to map one-to-one onto an actual bytecode. The `T1X` part that does the actual compilation, `T1XCompilation` extracts various information from the template definition via Java reflection, mostly regarding stack handling, e.g., for the `iadd()` method in the code given, the `@Slot()` annotations specify which argument is to be loaded with which stack slot. The return value of the `iadd` method specifies that a stack slot for an integer value is necessary and the value of the method will be pushed to the stack.

The simplified process of compiling a Java bytecode method is as follows:

1. The `T1X` compiler creates a new *compilation*, an object that represents the current process of compiling a method. That compilation holds the to-be-created native code as a buffer that can hold bytes. Also, a platform-specific

concrete subclass of the `T1XCompilation` has access to a limited assembler object that can write to the buffer.

2. The compilation writes a prologue to the buffer. This prologue contains typical low-level setup for a method entry, like adjusting the CPU stack pointer, putting the mehtod's arguments into the right place, or performing stack banging.
3. The compilation iterates over all the method's bytecodes.
   a) The template for the current bytecode is loaded.
   b) Depending on the `@Slot()` annotations of the arguments, the respective stack slots are linked to the argument registers for the template. This is necessary, as the `C1X` compiled templates use a register-based argument-passing scheme, as opposed to the stack-based argument-passing scheme of the Java bytecode. The respective native code is written to the buffer.
   c) Template arguments that lack a `@Slot()` annotation can now be filled in the compilation. In the example, the place for that code would be the `do_iadd()` method. Instead of the call to `emit()`, the method would contain a sequence of `start(TAG)`, methods that provide stack–register interaction like `peekInt(argument, name, stackPos)`, and `finish()`.
   d) The `finish()` method, which is implicitly called in the `emit()` method, then actually looks up the pre-compiled template and writes the native code to the buffer.
   e) Depending on the return type and annotations of the template method, the compilation emits code to push the result of the template to the stack.
4. At this point, all bytecode templates should be written to the buffer. Depending on the return type of the method to be compiled, special native code for returning is emitted to the buffer. Then, the compilation writes an epilogue, similar to the prologue, to the buffer.

The result of such a compilation is depicted in figure 6.5.

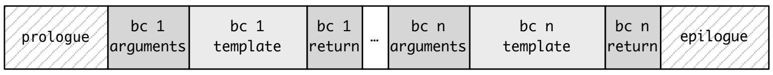


Figure 6.5.: Illustration of native code after compilation by `T1X`. Included are prologue and epilogue (stripe pattern), per-bytecode argument and return value handling code (dark gray) emitted by `T1X`, and the native template code (light gray) emitted by `C1X` before compilation.

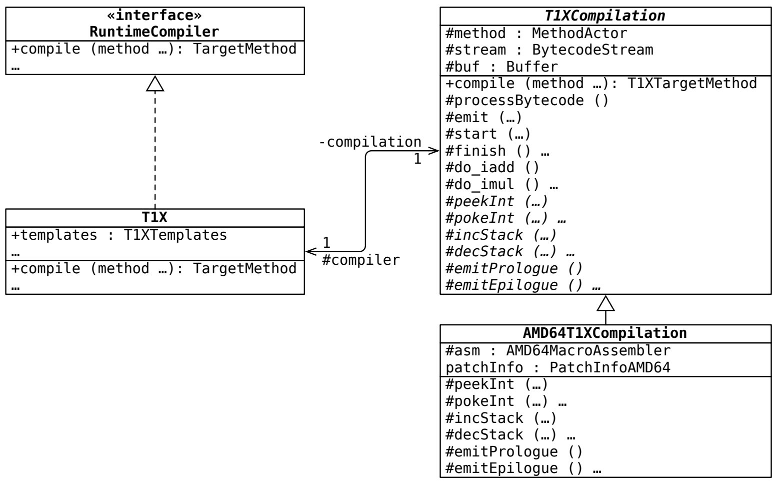


Figure 6.6.: `T1X` classes overview. A `T1X` compiler implements the `RuntimeCompiler` interface and holds onto the templates and the current *compilation* for a method that keeps track of the process of compiling a method.Key: UML class diagram.

Not all Java bytecodes can be expressed using `T1X` templates, first and foremost, stack modifying bytecodes and jump bytecodes. The former can be implemented without problems in a `do_*` method — stack modifying behavior is provided by platform-specific compilations and can just be used — , the latter, jump bytecodes, are however more challenging to implement. To allow for foreward jumps, `T1X` takes a record of jumps and their destinations and patches the native code accordingly after all bytecodes have been processed.

In any case, to obtain proper native code for any of the method's prologue, epilogue, stack modifying operations, or the just mentioned bytecode implementations that cannot use templates, `T1X` uses a platform-specific subclass of the `T1XCompilation` thst also implements the code patching mechanism mentioned. Such a subclass currently only exists for the AMD64 platform. The relations between the compilation classes and the compiler entry-point class `T1X` is depicted in figure 6.6.

In short, `T1X` is capable of generating native code from Java bytecode by means of pre-compiled code snippets and some glue code. Compared with `C1X`, this compiler is straightforward and easy to understand. The focus is an interpreter-compatible, complete implementation of the Java bytecode set.

⁂

To sum up Java bytecode compilation in Maxine, two bytecode compilers exist in Maxine, each with different but complementing intent. `C1X` is an optimizing bytecode compiler that aims for performance, `T1X` is a non-optimizing, template based bytecode compiler that aims to produce native code that is compatible with the original JVM interpreter. A compilation broker at runtime decides which compiler to use for a given method.

6.3.2. Squeak Compilation with `S1X`

Compiling Squeak methods in Maxine has been the major implementation effort for the execution part of SqueakMaxine. For this, we plugged into the compiler selection infrastructure and implemented a Squeak JIT. To keep the implementation as simple as possible we chose to implement a Squeak JIT by pursuing the approach of `T1X`, Maxine's Java baseline compiler. As SqueakMaxine does not change much of the VM infrastructure, the Java JITs are to stay in the VM and the JITs have to co-exist. Therefore, when a method is going to be compiled, it is necessary to know, which compiler to use.

Selecting the Right Compiler Maxine provides means for selecting compilers depending on a method's state. The broker mechanism outlined at the begin of section 6.3 can dispatch between optimizing and non-optimizing compilation for Java methods. Squeak methods can be handled by refining the broker. As depicted in figure 6.7, the Squeak compilation broker simply wraps the Java compilation broker and dispatches on the language of the method to be compiled. This information is shared in form of what subclass of `MethodActor` is to be compiled; the `SqueakCompilationBroker` only selects the Squeak compiler for instances of the `SqueakMethodActor` subclass. The current implementation of the baseline–optimizing selection requires that the same compiler is specified as first compiler and also as fail-over compiler. Hence the second compilation state with the Squeak compiler in figure 6.7 (dashed state). The outcome is known to be failure in any case when this state is reached.

The Squeak JIT `S1X` The one and only Squeak native code compiler available in SqueakMaxine is *`S1X`*, a template-based, non-optimizing JIT, modeled very closely after the `T1X` Java JIT — hence the name (*Squeak `T1X`*). In fact, it is so close that most of the infrastructure remains unchanged compared with

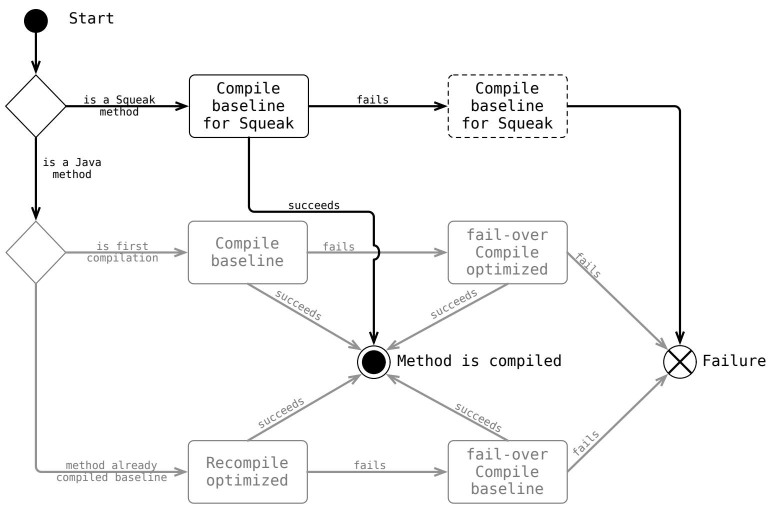


Figure 6.7.: Squeak compilation broker behavior. A Squeak method will be compiled with a Squeak compiler with no possibility of actual fail-over compilation (black part). A Java method is subject to the original broker behavior (gray part). Key: UML state diagram + dashing as explained.

`T1X`. Only the bytecodes set the compiler is working on and its interpretation is necessary to be altered. However, the changes to `S1X` compared to `T1X` go further than the bytecode set exchange. To implement the built-in behavior of Squeak (cf. section 6.5) in a simple way, we extended the template tagging system. Furthermore, in an attempt to unify `S1X` with `T1X`, `S1X` is able to handle more than one set of tags, which is `T1X` not. Yet, this subject to the compiler evaluation in section 8.2.

Find an example for a Squeak bytecode template in listing 6.2. Besides simple renaming and the different purpose of the bytecodes in the first place, the template-related code does not differ substantially from its `T1X` equivalent (cf. listing 6.1). Due to Squeak's object model `(Squeak)References` are prevalent as return types in template implementations, even if they denote boolean values, as in the example. Note the apparently strange construct of a **`final`** result variable that is set depending on the conditional and then returned. This construct is necessary to force the compiler compiling the templates — `C1X` — to create the proper assembly code for both outcomes, true or false. Without that workaround, i.e., the return statements directly in the branches, the compiler would generate assembly code that omitted the true-branch regardless of the conditionals value, a behavior specific to references.

```
/* SqueakTemplateSource part for PRIM_EQUIVALENT */
@S1X_TEMPLATE(value = PRIM_EQUIVALENT, primitive = 110)
public static SqueakReference primitiveEquivalent(
  @Slot(1) SqueakReference self,
  @Slot(0) SqueakReference other
) {
  final SqueakReference result;
  if (self.toOrigin().equals(other.toOrigin())) {
    result = trueReference();
  } else {
    result = falseReference();
  }
  return result;
}

/* S1XCompilation part for PRIM_EQUIVALENT */
protected void processBytecode(int opcode) throws InternalError {
  beginBytecode(opcode);
  switch (opcode) {
    case Bytecodes.PRIM_EQUIVALENT : do_prim_equivalent(); break;
    /* … */
  } /* … */
}

protected void do_prim_equivalent() { emit(PRIM_EQUIVALENT); }
```

Listing 6.2: `S1X` parts relevant for the `PRIM_EQUIVALENT` bytecode.

The `T1X` template system is originally laid out to support Java bytecodes in the sense that it is possible to denote certain methods as template implementation by adding an annotation to it. With the addition of `S1X`, more than one template-based compiler exist in SqueakMaxine. To introduce more flexibility in the template system, we add a new construct to dynamically determine which template annotation to use for the respective template compiler. In essence, the `T1XTemplateTagAdapter` provides this functionality by means of an interface of methods to access annotations of template methods and the template methods themselves. That way, it is possible to maintain different sets of template methods by using different template tags. We use this mechanism for the Squeak template methods. See listing 6.3 where the template tag for the class is specified by the `@T1X_SOURCE` annotation. This tag is then used for tagging the method `returnTopFromMethod` as template method. The `T1XTemplateTagAdapter` picks up the `@T1X_SOURCE` annotation and subsequently, `S1X` is able to determine the template tag when it is iterating over the classes methods via reflection. Hence, `S1X` is able to establish a connection between the bytecode (`RETURN_TOP_FROM_METHOD`) and the method that will implement that bytecode (`returnTopFromMethod`).

```
@T1X_SOURCE(S1X_TEMPLATE.class)
public class SqueakTemplateSource {
    /* ... */
    @S1X_TEMPLATE(RETURN_TOP_FROM_METHOD)
    @Slot(-1)
    public static Reference returnTopFromMethod(@Slot(0) Reference object) {
        return object;
    }
}
```

Listing 6.3: `SqueakTemplateSource` with variable template tag (`S1X_TEMPLATE`).

One major divergence from `T1X` is the implementation of branching behavior. Unconditional branching aside, Java's bytecode has a greater variety of branching instructions than Squeak's; the former supports equality and lower–greater comparison for all its primitive types, while the latter only knows about conditional branching based on a boolean on the top of the stack. The implementation for the Squeak branching logic is hence shorter and simpler. There are only four cases for Squeak:

1. Branch when the top of stack is the true-object.
2. Branch when the top of stack is not the true-object.
3. Branch when the top of stack is the false-object.
4. Branch when the top of stack is not the false-object.

Note that these cannot be combined as it may be the case that neither the true-object nor the false-object are the top-of-stack object. These boolean-object tests are mapped to either equality or non-equality native-code comparisons in the `AMD64S1XCompilation`, the low-level part of `S1X`. To find the right target native code to jump to, `S1X` uses the same patching approach as `T1X`. Additionally, the patching mechanism only supports jumps to bytecode boundaries, hence, the implementation of bytecodes that comprise multiple control flows is tricky when a template implementation does not fit and `do_*` methods have to be used.

When call instructions are compiled, unlike in `T1X`, method lookup is performed always at runtime due to the possibly polymorphic nature of the method to be called. This renders the optimizations present in `T1X` void that rely on the fact that certain methods can be looked up at compile time.

Apart from the functionality mentioned, the Squeak template compiler comprises the implementation for built-in behavior and access to runtime-information, covered in section 6.5.2 and section 6.6.2, respectively.

⁂

In short, basic bytecode compilation for Squeak with S1X uses the same approaches as bytecode compilation for Java with T1X, i. e., template-based non-optimizing compilation. Currently, there is no optimizing JIT for Squeak bytecode, a special broker exists to cope for that. S1X re-uses most parts of the T1X implementation. For reference, a complete compilation example for S1X is given in Appendix F, which already contains the handling of Squeak / Smalltalk primitives as explained in section 6.5.2.

6.4. Garbage Collection in the Presence of Executing Code

Garbage collection can happen during the execution of JIT-compiled code. However, not at any point but only so-called safe-points. Safe-points are native code operations that may result in a processor trap that Maxine can intercept. Under certain circumstances, the trap handling code for safe-points kicks of the GC. Usually, safe-points are generated right after calls, but may be present at other points in a native code method. Nevertheless, to perform proper clean-up, the GC needs a certain knowledge of the executing method's stack.

A *reference map* conveys the information of the current stack between JIT-compiled code and the GC. Such a map contains information about what parts of the stack contain what kind of data and, hence, information of what stack fields are references and therefore subject to garbage collection.

6.4.1. Reference Maps in Maxine

Reference maps for stack frames are subject to the nature of the frame's method. This nature, baseline or optimizing, has an impact on a frame's layout. This may result, e. g., in different frame sizes or different locations for method arguments or local variables.

C1X For C1X-compiled methods, reference maps are created from debug information the C1X compiler generates for each safe-point in a method. The type information for such methods stack frames is determined as early as the optimization process transforms the method to its lower-level intermediate representation (LIR) from.

`T1X` For `T1X`-compiled methods, every end of a bytecode might denote a safe-point.The baseline compilation behavior of `T1X` demands an interpreter-compatible stack behavior, i.e., when a bytecode, like the `dup` bytecode, results in an increase of the Java stack, `T1X` also has the frames stack increased after all native machine instructions belonging to the bytecode are executed. Moreover, the bytecode also determines the type of data put on or removed from the stack. Hence, a method's bytecode is sufficient to determine the reference map for each safe-point in a `T1X`-compiled method.

To create the mapping from bytecode indices, and, hence, safe-points, to reference maps, Maxine uses a `ReferenceMapInterpreter` that scans through the bytecode of a method and records *stack depth* and *data type* for each bytecode. Actually, not the data type itself is recorded but whether a stack slot contains a type that is subject to the GC, i.e., object references. Therefore, only a bit-map is necessary for reference maps.

6.4.2. Reference Maps in SqueakMaxine

The `S1X` JIT works very similar to `T1X` and hence, the handling of reference maps in SqueakMaxine is almost identical. Obviously, the bytecode set to be interpreted is different, however, the basic concept stays the same. In fact, the SqueakMaxine version of reference map interpretation is even simpler than that of `T1X`, as anything on the stack in SqueakMaxine is a reference, as every value in Squeak is an object. Therefore, the created reference bit-map is only dependent on the stack depth at a particular point of the execution.

6.5. Implementing Built-in Behavior

Besides the compilation of IRs to native machine code, the most important part of a languages execution model implementation is providing the languages built-in behavior (cf. section 4.4). However, while the implementations of JITs are quite similar, even across languages, the built-in behavior implementations might differ vastly. That said, all built-in behavior implementations share the notion of *intercepting* the normal method execution and *replacing* behavior by internal, predefined behavior. This notion suits at least both the Java and the Squeak built-in behavior; native methods as well as primitives replace calls to normal methods, regardless of whether they actually contained bytecode or not.

6.5.1. Maxine Java Method Substitution, Intrinsics, and the JDK

As a VM for Java, Maxine needs to provide the Java built-in behavior. This comprises the native methods of the JDK. However, the JDK is commonly deployed with implementations for these native methods in a binary, machine dependent version suitable for the default JVM. The JVM-implemented native methods of the JDK are actually code in libraries compiled from C++ code, like the standard JVM. These methods adhere to a certain calling convention, called the *Java native interface* (JNI). Maxine implements this calling convention for use with native methods. Hence, Maxine is able to re-use the JVM implementation for Java built-in behavior.

The default JVM implementation of most native methods in the JDK suits well for Maxine, too. However, some native methods contain code too specific to the standard JVM for being re-used by Maxine. This is especially true for methods that deal with reflection or, in some cases, implement object initialization. In order to re-use existing native method implementation on the one hand but provide own implementations of certain methods on the other, Maxine provides means to *substitute* selected methods of certain classes upon loading or compilation.

Substitution Maxine has means to intercept the default method lookup whenever certain methods are to be called. This technique is called *substitution* and is accomplished in the JIT of Maxine. As part of Maxine, a number of classes list methods to be substituted. These classes are annotated with `@METHOD_SUBSTITUTIONS`. All methods of these classes with a `@SUBSTITUTE` annotation are loaded by the compilers and themselves contain information, which method they should actually replace. This information is the name of the method, that together with the specially formatted name of the class identifies the to-be-substituted method. Whenever a compiler encounters a method that match any of the substitutes, the original implementation is just omitted and the Maxine provided one is used. In this substitution process, it is also possible to change or exchange whole classes or inject new fields into existing classes.

Intrinsics (outer) The generic technique of substituting methods is applied to and further refined for several crucial methods of the JDK, as explained before. These methods are called *intrinsics*. We call them *outer intrinsics* here to distinguish them from the intrinsics to be explained hereafter. Outer intrinsics are compiler-specific in that, depending on the compiler, the set of intrinsic methods might differ. For `C1X`, around 50 methods are marked as to be substituted as outer intrinsics.The `T1X` does not specify any outer intrinsics, as it makes use of `C1X` for its templates.

```java
package java.lang;
public class Object {

  public final native Class<?> getClass();
  public native int hashCode();
  protected native Object clone() throws CloneNotSupportedException;

  public final native void notify();
  public final native void notifyAll();
  public final native void wait(long paramLong) throws InterruptedException;
}
```

Listing 6.4: Native methods in `java.lang.Object`, extracted from the classfile.

For `C1X`, outer intrinsics are handled when the internal representation is transformed to its LIR form. Elements that match methods marked as intrinsics are treated specially. As an example, consider the reflective `getClass` method of `java.lang.Object`. To provide the desired functionality, i.e., that this method returns the correct Java `Class` object, `C1X` specifies an intrinsic. The Java `enum C1XIntrinsic` holds information which methods to intrinsify. Among other native methods of `java.lang.Object` (cf. listing 6.4), `getClass` is listed here (cf. listing 6.5, line 5).

Every intrinsic defined in `enum C1XIntrinsic` is specified with the method it should substitute by giving its class, name, and signature, as illustrated in listing 6.5. The `enum` holding such intrinsics also provides methods to determine, whether a given method is subject to a defined intrinsic (cf. line 9).

The intrinsification is carried out during JIT compilation. `C1X` builds a node graph representing the control flow in a bytecode method. While optimizing this graph, `C1X` tries to remove nodes representing a call to a method. During that phase, the to-be-called method is checked whether it is an intrinsic. If so, the call node is replaced by what the intrinsic specifies. In the case of the `getClass`, a pre-compiled piece of native code is inserted that determines the Maxine actor for the class of the object `getClass` is called upon and returns the according Java class object.

```
public enum C1XIntrinsic {
  // java.lang.Object
  java_lang_Object$init("java.lang.Object", "<init>", "()V"),
  java_lang_Object$hashCode("java.lang.Object", "hashCode", "()I"),
  java_lang_Object$getClass("java.lang.Object", "getClass", "()Ljava/lang/Class;"),
  java_lang_Object$clone("java.lang.Object", "clone", "()Ljava/lang/Object;"),

  C1XIntrinsic(String className, String methodName, String signature) { /* ... */ }
  public static C1XIntrinsic getIntrinsic(RiResolvedMethod method) { /* ... */ }
}
```

Listing 6.5: Excerpt form `enum C1XIntrinsic`: The intrinsics' definitions for `java.lang.Object`

Intrinsics (inner) Another concept referred to as intrinsics in Maxine are the constructs that make Maxine a superset of Java (cf. section 2.1.1). While not necessary to explain the implementation of Java's built-in behavior in Maxine, we give a very brief explanation of these intrinsics. We will refer to them as *inner intrinsics* to differentiate them from the outer intrinsics explained previously. To achieve the superset-of-Java behavior, e.g., the `Word` data type being mapped to actual machine words, source code expressed in standard Java has to be treated specially. Maxine utilizes annotations and a special class hierarchy for this purpose. For the latter, `Word` data types, instances of a `Word` subclass are compiled and inlined in such a way that only machine-specific numbers remain, e.g., while `Address.zero()` returns an instance of `Address` at the Java level, Maxine inlines and compiles this to the bit-pattern representing the number `0` for the current native instruction set. This poses a special case of substitution.

The actual inner intrinsics are a set of less than 20 operations that can be invoked by giving methods a special annotation. The `UNSAFE_CAST` operation is a principal example of Maxine inner intrinsics. Calls to methods with such an annotation do not compile to any native code but are useful to circumvent restrictions of the Java type system, where certain assignments are forbidden. Other inner intrinsics include operations for memory manipulation, allocation, or manual trapping. This set is defined in the `MaxineIntrinsicIDs` class and both `C1X` and `T1X` provide implementations for them. Note that the latter is a template-based implementation very much like the bytecode implementations described in section 6.3.1.

Table 6.1.: Differences between bytecode and primitive behavior for functionality provided by either.

	bytecodes	primitives
Scope	within the invoking method	outside of the invoking method
Needs new stack-frame	✘	✔
Failure handling	continue with a message send	continue with the body of the containing method
	Example: Equality test	
Name	bytecodePrimEqual	primitiveEqual
Number	183	7
Successful outcome	`true` or `false` on stack[a]	`true` or `false` on stack
Failure outcome	resend of `#=` at bytecode level	invocation of method body of `SmallInteger>>#=`

[a] The Squeak interpreter VM performs an optimization within the bytecode by peeking into the next bytecode to see if it was a conditional jump, and if so, directly performs it.

6.5.2. SqueakMaxine Primitives

Contrasting Java, Squeak's built-in behavior is commonly implemented by the VM and not the class library as it is the case with the JDK and Maxine. A subset of the essential primitives (cf. section 4.1.3) is available to SqueakMaxine through an extension of the S1X template compiler.

Squeak primitives are handled at the same time method bytecodes are handled, i.e., within the JIT. A Smalltalk method essentially consists of two parts, bytecode or primitive number, of which at least one is necessary. From the perspective of the call site, it is irrelevant, whether a primitive, bytecode, or both is executed. Hence, providing primitive handling at the same point as bytecode handling is possible; for many bytecodes there is a corresponding primitive and vice versa. While they differ in their invocation scope and their outcome in exceptional cases, they share most of their functionality. The main differences between both are given in table 6.1.

Seeing that several bytecodes and primitives share the same functionality, we chose to use the same means to implement primitive behavior as we use to implement bytecode behavior, i.e., the template system of S1X. Hence, S1X templates are responsible for both bytecode and primitive implementation.

In essence, a method's primitive is handled as if it were a special bytecode before all other bytecodes. Since the primitive number in is just `0` for methods without a primitive, the simplified compilation process for a nmethod is roughly equivalent to the steps given in section 6.3.1, but with the additional primitive step.

1. Emit a method prologue
2. If the method has a primitive number greater than `0`, handle the primitive:
   a) Emit its corresponding native code.
   b) Emit a fail check
   c) Emit a branch to the actual bytecode in case primitive fails.
   d) Emit epilogue that ends the method and returns the result.
3. For each individual bytecode
   a) Emit code for the bytecode
4. Emit an epilogue that ends the method and returns the result.

The aim to reuse the templates for bytecodes and primitives as well effected an extension of the `T1X` template system. It is possible to specify primitive numbers in addition to bytecode tag names in a templates definition. Consider listing 6.2 that gives the relevant parts for implementing the bytecode `primitiveEquivalent`, and compare it to listing 6.6. The template — and hence the resulting machine code — is the same, but it is emitted to the final native method at a different point and guarded with a fail check.

An extended template method has to meet the following criteria:

Type The method must be a **`public static`** method.

Tag The method must bear a `@S1X_TEMPLATE` annotation tag.

Tag name The tag must contain a bytecode tag or `PRIMITIVE`.

Primitive number The tag may contain a primitive number or a list thereof.

Slot annotations The method and its arguments may bear `@Slot` annotations as described for `T1X` (cf. section 6.3.1). A template method for a primitive must include a `@Slot(-1)` annotation.

Consider for example the following three templates.

```
@S1X_TEMPLATE(RETURN_RECEIVER)
@Slot(-1)
public static SqueakReference returnReceiver(...) {...}
```

This method specifies a template for a single bytecode.

```
@S1X_TEMPLATE(value = PRIM_EQUIVALENT, primitive = 110)
public static SqueakReference primitiveEquivalent(...) {...}
```

This method specifies a template for both a bytecode and a primitive.

```
@S1X_TEMPLATE(value = PRIMITIVE, primitive = {130, 131})
public static SqueakReference garbageCollect() {...}
```

Finally, this method specifies a pure primitive template for two primitives.

```
/* SqueakTemplateSource part for primitive 110 */
@S1X_TEMPLATE(value = PRIM_EQUIVALENT, primitive = 110)
public static SqueakReference primitiveEquivalent(
    @Slot(1) SqueakReference self,
    @Slot(0) SqueakReference other
) {
    final SqueakReference result;
    if (self.toOrigin().equals(other.toOrigin())) {
        result = trueReference();
    } else {
        result = falseReference();
    }
    return result;
}

/* S1XCompilation part for primitive 110, using the general primitive case */
protected void processPrimitive(int primitiveIndex) throws InternalError {
    switch (primitiveIndex) { /* … */
        default: emitPrimitive(primitiveIndex); break;
    }
}

protected void emitPrimitive(int primitiveIndex) {
    startPrimitive(primitiveIndex);
    // … (load template, assign possible result)
    finish();
    // branch to actual bytecode (index 1) if primitive failed
}
```

Listing 6.6: S1X parts relevant for the Squeak primitive 110.

Failure handling A Squeak primitive might fail. This can happen, e.g., when a primitive's arguments are not of the expected class, a computation effected an exceptional result like a division by zero, a primitive is not provided by the VM, or even intentional. In any case, when a primitive fails, the primitive's method's body is to be executed. To achieve this, it is necessary to provide an indication of failure in the JIT compiler. Moreover, the method body is to be executed with the original arguments, hence, they are to be restored if changed by the primitive in any fashion.

As some templates are shared between bytecodes and primitives, introducing a global primitive success flag as the original Squeak VM does is undesirable, and even more so since Maxine provides native threads opposed to Squeak's green threads. Therefore, S1X templates now follow the convention to result in a `null` value in case of a failure. This value can be tested and it is possible to either return from the primitive's method in case of success (line 9 of listing 6.7) or continue with the method's body in case of failure. This is done by the branching in line 10 of listing 6.7.

```
 1 protected void emitPrimitive(int primitiveIndex) {
 2   startPrimitive(primitiveIndex);
 3   int stackArgs = template.sig.stackArgs;
 4   CiRegister result = template.sig.out.reg;
 5   finish();
 6   if (stackArgs > 0) {
 7       incStack(stackArgs); // unpop
 8   }
 9   branchIfNull(result, 1, 0);
10   emitEpilogue();
11 }
```

Listing 6.7: Squeak primitive failure handling by jumping over the primitive return.

To meet the requirement of restoring the method's original arguments, `S1X` just "un-pops" the stack. This is possible, as popping the stack in `S1X` essentially decrements the stack pointer without actually modifying the stack top value. Furthermore, in recent Squeak versions, it is forbidden to overwrite a methods arguments, hence, they remain untouched on the stack. The immediate result of a `S1X` template is stored in a register, given the template bears a `@Slot(-1)` annotation. Hence the last requirement in the template criteria list above. Consequently, only the original arguments reside over the stack and can be un-popped as done in line 6 et seqq.

For SqueakMaxine, a subset of the Squeak primitives is implemented; not all essential primitives are implemented but a few non-essential ones are. Please refer to table D.1 in Appendix D for a full list of supported primitives. There is an exclusive choice between primitives supporting traditional Smalltalk-80 contexts and newer closures. While both sets are marked essential, only one set is actually necessary; SqueakMaxine implements the closure primitives. A number of primitives are expected to fail. These 76 primitives are implemented using a template that just returns **`null`**, thus employing the already outlined failure handling. This is also indicated in the last line of the said table. Similar to bytecodes, primitives can be implemented at different levels within Maxine and, more specifically, `S1X`. Most primitives are implemented with a template. See table 6.2 for the implementation level of primitives implemented by SqueakMaxine. Some templates need functionality, such as allocation, that potentially result in a compiler stub—small, handcrafted native machine code for special purposes internal to Maxine—, which is not possible for templates. Hence, the `S1XRuntime` provides the requested functionality, marked as not to be inlined, and the respective template can call the runtime functionality safely.

Table 6.2.: Primitives and their implementation levels.

		Implementation level		
Index	Name	Compilation	Template	Runtime
1	Add		✔	
2	Subtract		✔	
3	LessThan		✔	
4	GreaterThan		✔	
5	LessOrEqual		✔	
6	GreaterOrEqual		✔	
7	Equal		✔	
8	NotEqual		✔	
9	Multiply		✔	
10	Divide		✔ [a]	
12	Div		✔	
17	BitShift		✔	
60	At		✔	✔
70	New		✔	✔
71	NewWithArg		✔	✔
110	Identical		✔	
111	Class		[b]	✔
130/131	Full/IncrementalGC		[b]	✔
201–205	ClosureValue	✔	✔	✔
256	Quick return receiver	✔		
257	Quick return `true`	✔		
258	Quick return `false`	✔		
259	Quick return `nil`	✔		
260–263	Quick return −1 – 2	✔		

[a] Fails, the image side code is sufficient. [b] Call to the runtime method.

Besides normal primitives, Squeak allows for special primitives for fast access to common objects or instance variables, called *Quick primitives*. SqueakMaxine provides a limited range of quick primitives, i.e., those listed as "Quick return …" in table 6.2; quick primitives that return instance variables are not implemented. The eight quick primitives that are available are not implemented at template level, as they are very close to their respective bytecodes, which are implemented at compilation level. Quick primitives are prevalent in Squeak, there are around 1,600 methods that use the quick return primitives as given in table 6.2, an more than 3,600 methods altogether using quick primitives in a vanilla Squeak 4.3 image, but these numbers might differ vastly, as methods using these primitives may be created at will by the image-side Squeak bytecode compiler when a method solely consists of a return of the receiver, **`true`**, **`false`**, **`nil`**, a number between -1 and 2, or an instance variable of an object. The number of those methods differs from application to application.

⁂

Summing up built-in behavior implementation, Maxine does not implement the Java built-in behavior but reuses the JDK implementation, with means to substitute behavior if necessary. Conversely, Squeak primitives are handled like bytecodes in SqueakMaxine, and the existing `S1X` is reused for the implementation of Squeak's built-in behavior.

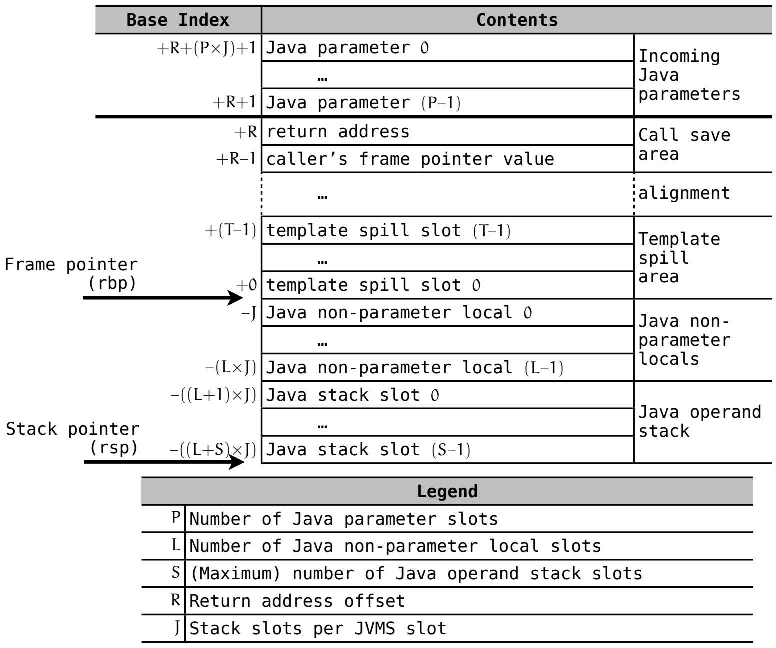
6.6. Runtime Execution Information and Call Stacks

For debugging and meta-programming purposes, it may become necessary to provide information about the current execution. First and foremost, such information can be *stack traces*, mostly a textual representation of the current call stack, especially for Java. However, Squeak allows an object-oriented access to the call stack, which has to be reflected in SqueakMaxine. In any case, Maxine has to keep track of the current execution and is to provide access to it.

6.6.1. Stack-walking in Maxine

The access to runtime execution information in Java is practically limited to stack traces, eventually accessible though the methods `getOurStackTrace()` and `fillInStackTrace()` of the class `java.lang.Throwable`; and limited reflective access to a methods caller using `getCallerClass()` of the reflection class `sun.reflect.Reflection` as provided by the JDK. Maxine just has to provide customized implementations for these methods in the sense of section 6.5.1. However, internally, more than just full stack traces are necessary, and hence Maxine provides a more elaborate means of stack access, called *stack walkers*.

Maxine stack walkers are a means to iterate over the elements of a stack. They provide *stack frame visitors* that can be provided to do actual work on the stack. Thus, the implementation for a full stack trace is a visitor that collects the visited stack frames as `java.lang.StackTraceElement` objects. A stack trace printer is even simpler, as it just needs a stack frame visitor with a single logging statement.



Base Index	Contents	
+R+(P×J)+1	Java parameter 0	Incoming Java parameters
	…	
+R+1	Java parameter (P−1)	
+R	return address	Call save area
+R−1	caller's frame pointer value	
	…	alignment
+(T−1)	template spill slot (T−1)	Template spill area
	…	
+0	template spill slot 0	
−J	Java non-parameter local 0	Java non-parameter locals
	…	
−(L×J)	Java non-parameter local (L−1)	
−((L+1)×J)	Java stack slot 0	Java operand stack
	…	
−((L+S)×J)	Java stack slot (S−1)	

Legend	
P	Number of Java parameter slots
L	Number of Java non-parameter local slots
S	(Maximum) number of Java operand stack slots
R	Return address offset
J	Stack slots per JVMS slot

Figure 6.8.: `T1X` frame layout for AMD64 architectures, based on the Java virtual machine specification. Source: `AMD64JVMSFrameLayout`

A stack frame walker, as implemented by `StackFrameWalker`, calls the stack frame visitor it is given with appropriate values describing the current frame, i.e., data structures that hold information like, e.g., the current stack pointer, frame pointer, or instruction pointer. These values are *read only*.

Currently, there are two stack frame visitor base classes in Maxine, the `StackFrameVisitor` and the `RawStackFrameVisitor`. While the former provides a more object-oriented view on the stack, it allocates new objects during the walking process and hence may not suit all situations where stack traces are necessary. Therefore the latter provides an allocation-free, yet more machine-oriented view on the stack. Apart from the Maxine inspector, only the `RawStackFrameVisitor` is currently used in the Maxine VM implementation, due to its non-allocating behavior. Stack frame visitors are chiefly used for debug information, and information gathering and stack patching for both de-optimization [19] and code eviction, among others.

However, from the Java perspective only little of this powerful interface is necessary. Hence it is possible for Maxine to provide the necessary information by substituting the indicated methods (e.g., `getOurStackTrace()`,

```
Generator>>fork
  | result |
  home := thisContext.
  block reentrant value: self.
  thisContext swapSender: continue.
  result := next.
  continue := next := home := nil.
  ↑ result
```

Listing 6.8: Example use of `thisContext`. The active context is stored in an instance variable and receives a message.

`fillInStackTrace()` of `java.lang.Throwable`) with a stack walker based implementation suiting Maxine. The same holds for `getCallerClass()` of the class `sun.reflect.Reflection`. Nothing of the more powerful functionality of Maxine stack walkers, let alone write access, is exposed to Java.

6.6.2. SqueakMaxine and Smalltalk's `thisContext`

Smalltalk is renowned for its reflection and meta-programming capabilities. This includes static meta-programming, like class lookup by name or class creation at runtime, which affects or reflects on structural parts of Smalltalk programs. Moreover, Smalltalk supports dynamic meta-programming, e.g., intercession of or introspection into running methods and the call stack. To access such execution information, Smalltalk has a notion of *contexts*. A context is defined as the execution state of a method [14], including information about the calling method. Thus, the call stack in Smalltalk is represented by a linked list of contexts. In Smalltalk source code, the current method's context is represented by the pseudo-variable **`thisContext`**. While being handled specially with respect to certain message sends, Smalltalk contexts remain objects just as any other Smalltalk object. Hence, execution state can be saved to instance variables of objects and multiple stack paths are possible. The latter, as Deutsch [10] points out, is similar to *spaghetti stacks* of Interlisp. That way, concepts like coroutines [9], continuations [47], or generators can be implemented without special VM support. For example, consider a part of Squeak's generator implementation in listing 6.8, where the current context is accessed both for reading and writing.

In the traditional Squeak interpreter VM, the Smalltalk stack is modeled precisely after the context notion; the image side context objects, i.e., instances of `ContextPart`, are identical to the VM side context that make up the Smalltalk stack. Any new activation hence results in the creation of a new context object. This has positive implications, e.g., the stack practically being only limited by the memory available, but also downsides, e.g., the more difficult detection of infinite recursion or the necessary allocations.

Context reification The direct context mapping approach of the Squeak interpreter VM is simple but impedes performance and hinders efficient implementations of JITs. A possible solution for this is to maintain both a context-based Smalltalk stack and a VM internal call stack. This approach implies a rather high memory consumption and requires permanent synchronization between the two stacks, which is quite expensive. For this reason, several Smalltalk implementations, e.g., VisualWorks [27] or, for Squeak, the *stack interpreter VM* and the *CogVM* [28], do not create actual context objects for every activation any longer but internally use a C like stack and create context objects only when necessary. This *context reification* provides the best of both approaches, image side access to contexts and spaghetti stack on the one hand and VM side C-like stack for saving allocations and efficient access from JITs on the other hand. Yet, an implementation using this idea has to consider that image side contexts and VM side stack frames have to be sync, that created contexts must be stored appropriately, and that a context may outlive the stack frame it was created for.

On his CogVM website, Miranda proposed [29] a nomenclature for the states of a stack-frame–context relation as follows:

SINGLE A single context is a context that has no corresponding stack frame.

MARRIED A married context is a context object that has an associated stack frame. The married context acts as a proxy to the frame. Frames and contexts are married whenever a context is requested by image side code, i.e., "creating a block, explicit use of **`thisContext`**, [and] access to sender when sender is a frame […]" as stated in source code of Miranda's CogVM and Stack interpreter VM. The context object and the stack frame have to be kept in sync.

WIDOWED A widowed context is a context object that *is bereaved* of its stack frame, i.e., the stack frame belonged to a method that has already returned.

Both Miranda's stack interpreter VM and the CogVM provide this tri-state and the appropriate synchronization. In essence, the VMs intercept accesses to a context's instance variables and act on the frame accordingly. Conversely, the VMs' stack frames have a designated field for a married context, to provide the back-link (cf. figure 6.9). The state of a context–frame relationship

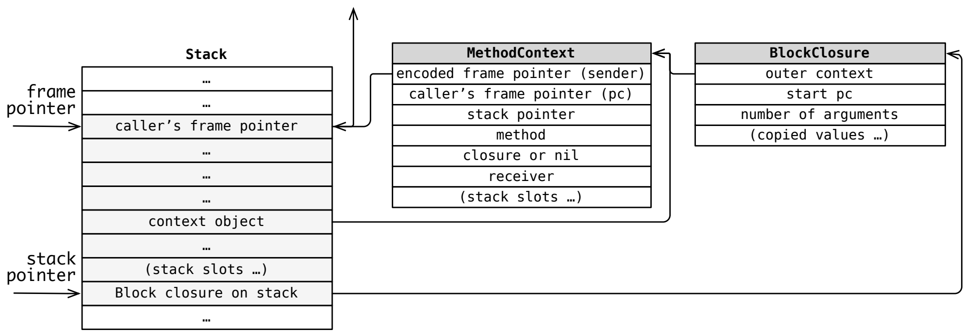


Figure 6.9.: Married contexts. The current stack frame (light gray) is "married" to a context object (`MethodContext`) that is an outer context for a closure (`BlockClosure`). The link from the context object to the stack frame is encoded into the context's *sender* and *pc* fields, which are treated specially by the VM. Adapted from an image by Miranda [29].

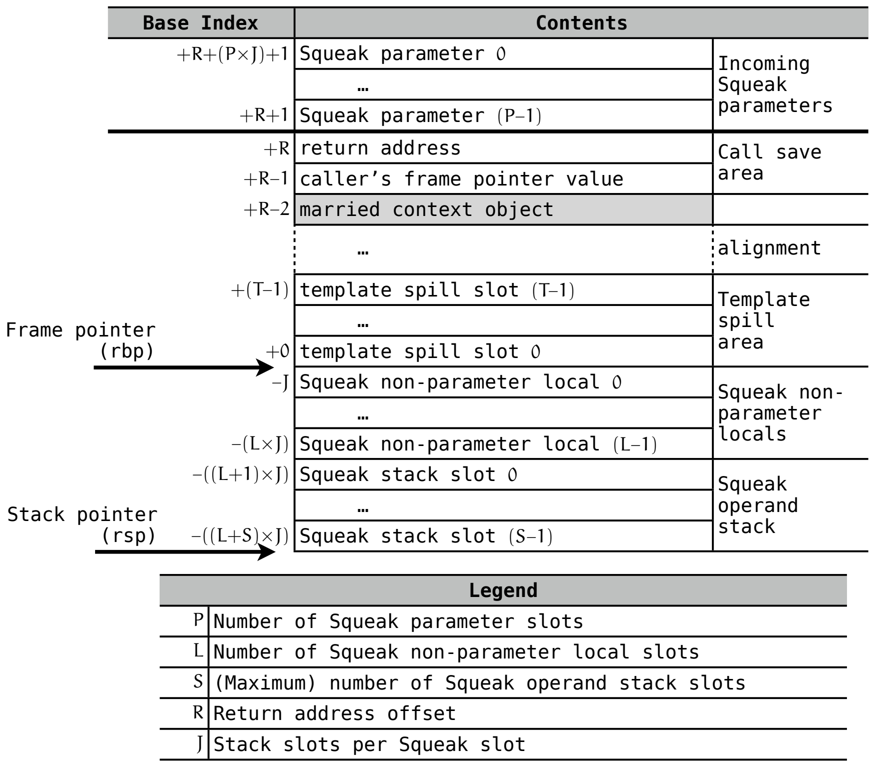
hence can be derived from the values of the context's instances variables and the frame's context fields, of which at least one must exists.

SqueakMaxine pursues the notion of context reification just like Miranda's VMs, yet, with slight adaptions to the Maxine VM.

Frame modifications To both implement the married context approach as well as remain as close as possible to the `T1X` implementation of Maxine, we modified the frame layout as used by `T1X` to include an optional *context* field, as depicted in figure 6.10. Furthermore, the `S1XCompilation` hast to provide the special handling of the instance variables of a context object. As all methods that access instance variables of context objects are compiled specially, only a few bytecodes are to be adapted to intercept access to context objects, namely the push- and store-instance variable bytecodes that use a prefix.

Most important, however, is the bytecode requesting the active context, `pushActiveContext`. This is the key interface between the Squeak image and the VM for context access. Basically, the following is performed when the active context is requested:

1. Check whether the current frame already has a context object. If so, the creation of a new object is skipped and the processing continues with step 4.
2. If not, a new context for the current frame is created:
   a) The current state of imported registers, i.e., frame-, instruction-, and stack-pointer, is recorded now. This happens via a template.



Base Index	Contents	
+R+(P×J)+1	Squeak parameter 0	Incoming Squeak parameters
	…	
+R+1	Squeak parameter (P–1)	
+R	return address	Call save area
+R–1	caller's frame pointer value	
+R–2	married context object	
	…	alignment
+(T–1)	template spill slot (T–1)	Template spill area
	…	
+0	template spill slot 0	
–J	Squeak non-parameter local 0	Squeak non-parameter locals
	…	
–(L×J)	Squeak non-parameter local (L–1)	
–((L+1)×J)	Squeak stack slot 0	Squeak operand stack
	…	
–((L+S)×J)	Squeak stack slot (S–1)	

Legend	
P	Number of Squeak parameter slots
L	Number of Squeak non-parameter local slots
S	(Maximum) number of Squeak operand stack slots
R	Return address offset
J	Stack slots per Squeak slot

Figure 6.10.: `S1X` frame layout based on the JVMS layout used by `T1X`. The only difference is the additional field before the call save area for the context object.

b) As a new object has to be allocated now, the template code calls the runtime with the pieces of information that describe the current stack frame, notably
   - the three said registers,
   - the method object (`CompiledMethod`) of the stack frame's method,
   - the closure object of the current stack frame, if applicable, and
   - the receiver, on which this method is called.

c) The `S1XRuntime` allocates a new context object and fills it with the supplied information from the template code. However, as with Miranda's VMs, the frame pointer is encoded into the *sender* field of the object. This tells subsequent accessors of the context that all three registers are stored in this context and the thusly shadowed *stackPointer*, *sender*, and *pc* values are to be derived on demand.

d) The new context object is handed back.

3. The new context object is saved to the *context* field of the current frame.
4. The context object is now filled with the current local variables.
5. The context object is put on the stack.

As pointed out, the access to instance variables of context objects has to be treated specially. This is necessary to obtain actual runtime information, e. g., when sending **`thisContext`** `pc`, the outcome depends on where in a method or closure this is sent. Hence, it is not sufficient to fill the instance variables of a context once and for all but they must rather be updated upon every call. As shown in listing 6.9 and 6.10, instance variable access for context objects is compiled to prefixed Smalltalk bytecodes, thus making it rather easy to do a class check upon every prefixed bytecode invocation. Had the `#pc` method been compiled with a non-prefixed bytecode, i.e., `pushReceiverVariable02` in this case, a class check would decrease performance heavily.

```
pc
  "Answer the index of the next
   bytecode to be executed."

  ↑pc
```

Listing 6.9: `ContextPart>>pc` source code

```
13 <84 40 01> pushRcvr: 1
16 <7C> returnTop
```

Listing 6.10: `ContextPart>>pc` bytecodes. Note the use of the extended bytecode.

The `S1X` compiler handles extended bytecodes and for the context sensitive bytecodes, it does a class check. Manual jumps in the `S1X` infrastructure are only possible with bytecode boundaries as target, hence we chose to unconditionally execute the normal bytecode behavior and do a class check after that, as can be found in listing 6.11, line 3 for the prefixed push bytecode. Note that the result of the normal, non-context operation remains on the stack. In the context case, and only if the context is married, this result is replaced with an actual value.

```
1 protected void do_push_receiver_variable_maybe_context(int index) {
2   do_push_receiver_variable(index);
3   do_check_context_or_next_bc();
4   // handle context
5   do_push_context_variable(index);
6 }
```

Listing 6.11: Implementation of pushing instance variable, prefixed variant.

Stack walking implementation Access to context instance variables, such as the prefixed push instance variable, requires access to the current stack frame and the associated previous stack frames. This is important to support access to the sender of a message. Hence, `S1X` uses the stack walking facilities of Maxine to access all necessary data. Consider the simple message send **`thisContext`** `sender`. When compiled, this code will eventually call the `S1XRuntime` method `variableInContext()`. Its parts that are relevant for this example are depicted in listing 6.12. Note that at this point, it is already guaranteed that the frame is married, thus line 2 will yield the frame pointer for the context.

```
 1 public static SqueakReference variableInContext(SqueakReference context, int index) {
 2   Pointer spouseFP = frameOfMarriedContext(context);
 3   Pointer spouseSP = r().getInstanceVariable(context, STACK_POINTER).toOrigin();
 4   Pointer spouseIP = r().getInstanceVariable(context, INSTRUCTION_POINTER).toOrigin();
 5
 6   if (index == ContextFieldIndex.SENDER.getIndex()) {
 7     ensureCallerContext(spouseIP, spouseSP, spouseFP);
 8     return getCallerContext(spouseIP, spouseSP, spouseFP);
 9   } …
10 }
```

Listing 6.12: Parts of `S1XRuntime.variableInContext()` that are relevant for determining a context's sender

For the determination of the sender context, `S1X` first ensures that a context in the caller frame exists and then retrieves it. Both operate using Maxine's stack walking. The method `ensureCallerContext()` just kicks off the stack walking using an `EnsureCallerContextStackVisitor`. It will walk the stack until the second-to-topmost frame and check for an existing or create a new context object for that frame. In listing 6.13, this visitor is given. A previously created stack walker uses this visitor and performs the walking with the important registers (stack, frame, instruction pointer), as determined in listing 6.12.

The method `getCallerContext()` essentially works the same way, with the notable difference that the context object that is found in the caller frame is stored locally in the visitor used. That way, `getCallerContext()` can retrieve it *after* the stack walk, as returning a value from a stack walk is not possible. Retrieving the Squeak instruction pointer works similarly; determining the stack depth is even simpler, as it only depends on the already known stack pointer and instruction pointer.

```
private static class EnsureCallerContextStackVisitor extends StackFrameVisitor {
  @Override
  @NO_SAFEPOINT_POLLS("Working on references")
  public boolean visitFrame(StackFrame stackFrame) {
    if (stackFrame instanceof SqueakFrame) {
      SqueakFrame sf = (SqueakFrame) stackFrame;
      if (!sf.frameHasContext()) {
        final S1XTargetMethod s1xMethod = (S1XTargetMethod) stackFrame.targetMethod();
        final SqueakReference receiver = SqueakReference.fromReference(
          sf.localsPointer(0).readReference(Offset.zero()));

        final Object newCtx = createContextRuntime(
          SqueakReference.fromJava(s1xMethod.squeakMethodActor().getCompiledMethod()),
          SqueakReference.fromOrigin(sf.fp.plus(1)), // is: sender,
          SqueakReference.fromOrigin(sf.ip),
          SqueakReference.fromOrigin(sf.sp),
          nilReference(),
          receiver);
        sf.frameContextPointer().writeReference(Offset.zero(),
                                               Reference.fromJava(newCtx));
      }

      if (sf.isTopFrame()) {
          return true;
      }
    }
    return false;
  }
}
```

Listing 6.13: `EnsureCallerContextStackVisitor`: Visitor to ensure that the caller frame is married. If not, a new context for that frame is created.

Using the stack walking approach, S1X only creates context objects when actually necessary and actually provides context reification.

7. Variation Points of a Virtual Machine Framework

When evaluating the extensibility, the final goal of using Maxine as a framework for virtual machines of other languages can be understood in two ways:

On the one hand, Maxine's source code can be reused as a starting point for the development of VMs for other languages. The additional code is integrated into Maxine during the bootstrapping phase, which will produce a customized VM capable of running both Java and a second language. This VM can then be shipped as an individual program.

On the other hand, Maxine could be regarded as a platform capable of executing multiple languages. In this scenario, a custom VM is but a Java program, with Maxine specific dependencies, which will be loaded by Maxine with some simple kind of extension mechanism. Bootstrapping the VM-extension will not be necessary as Maxine is already capable of loading and compiling Java code. This simplifies the development process as well as the distribution of the final program, because only one platform-independent jar file, containing only the additional sources, has to be provided.

SqueakMaxine was developed using the first approach, as it was necessary to modify some of Maxine's internals and no plug-in mechanism exists yet. In the long run, however, it might be desirable to have support for the second approach as well, as it might even allow running multiple language environments at the same time.

The remainder of this chapter will discuss how Maxine's extensibility can be improved, especially to be more flexible towards support other object models, regardless of which of the two framework approaches will be pursued. After a general analysis on the introduction of new language environments, missing variation points that have been discovered during the course of this project will be discussed for Maxine's subsystems.

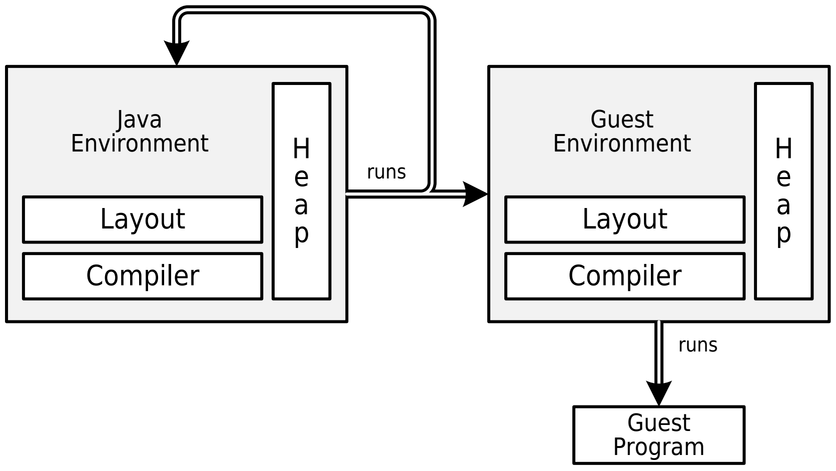


Figure 7.1.: Parallel environments. The host VM is used to execute the guest VM.

7.1. From Configurability to Extensibility

Maxine was started as a research Java VM that allows replacing subsystems easily to try out different implementations of aspects of the Java programming language. With the introduction of a new language, however, the existing parts must not be replaced, as Maxine requires them to run itself. Instead, new concepts have to be added without breaking support for Java. This can be done in two ways.

7.1.1. Parallel Environments

SqueakMaxine's first layout implementation, the Squeak default layout, was incompatible with the existing Java layouts. The main reason for this was the absence of a hub and a misc header field, which are heavily relied upon in Maxine's default implementation. As a result, Squeak and Java objects could not be stored on the same heap, nor could they reference each other. The Java heap, layout and compiler were just an implementation detail of the SqueakMaxine VM, which again implemented a heap, layout and compiler for Squeak.

As illustrated in figure 7.1, this scenario contains two independent language environments. The Java environment, essentially an unmodified Maxine, runs a guest environment and itself. SqueakMaxine, shown as the guest environment, is run by the Java part and executes the Squeak program.

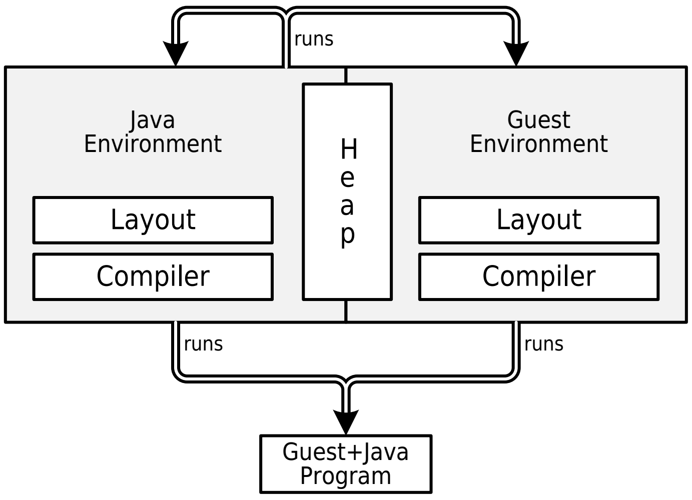


Figure 7.2.: Integrated environments. Both host and guest VM share components and can execute a program mixed of both languages.

7.1.2. Integrated Environments

With the second layout implementation, SqueakMaxine provides a fully integrated solution. All Squeak objects can be used wherever Java objects are expected, which allows having Squeak and Java objects on the same heap, referencing each other.

This approach, as shown in figure 7.2, still allows for a language specific layout and compiler. There are, however, some constraints the object model implementation has to fulfill to ensure compatibility:

Header Fields The hub and misc header fields have to be at the same offset in both layouts.

Origin in Cell The conversion between cell pointer and origin pointer of the object has to be possible without knowing the specific layout.

Meta-objects The hub has to reference an appropriate `SpecificLayout` implementation. To allow reflection in Java, fields and methods have to be represented with their respective actors.

Method Invocation and Field Access For a full integration, method invocations and field accesses have to be implemented in a way that also works for Java objects. On the other hand, fields and methods of the guest language don't have to be accessible to Java. As Java is statically typed, it is not possible invoke methods or access fields that are not available to the Java compiler anyway.

The two latter constraints might even become unnecessary with the introduction of wrapper objects which encapsulate the logic for using objects of another language or object model, but this is out of the scope of this work and requires further research.

7.1.3. Scheme Sets

It should be noted that both figure 7.1 and figure 7.2 show an idealized architecture of their respective approach that was not actually achieved. At the time of the project Maxine only allowed one instance of each scheme, which is globally accessible. Thus, the schemes had to be split to provide two implementations at once.

For the first approach, implementing Squeak's native layout, this worked well for the heap (see variation point "Garbage Strategy"), but not so well for the layout. A possible solution to this problem is discussed in variation point "General Layout Responsibility".

With the second approach, reusing and extending Maxine's default layout, it was not necessary to change the functionality of the heap. The modified layout implementation fulfilled the compatibility constraints as listed above. Thus, after providing the variation points Layout Responsibility and Reference Visitor, it was possible to add Squeak specific layouts without affecting Java objects.

This shows that the understanding of a scheme configuration has to be changed. Both approaches have in common that schemes are no longer globally unique, but that the relevant implementation depends on the context where it is used. For Maxine to be fully extensible, the global accessors for schemes have to be replaced with some local environment that provides one implementation for each scheme interface. Optionally, this mechanism can allow to introduce entirely new scheme interfaces to use the existing configurability mechanism to try out different implementations for aspects of the guest VM that are not present in Java/Maxine.

A helpful framework is not only easily extensible, but should also have reusable parts, like a library. To avoid forcing the reinvention of the wheel, it should provide meaningful default implementations for the defined extension points which can be adapted by customizing well defined variation points. The next three sections will discuss missing variation points that were identified during the implementation of the Squeak object model, ordered by their respective Maxine subsystem.

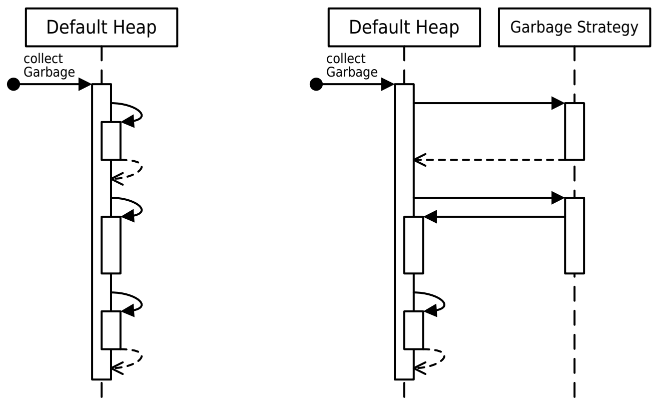


Figure 7.3.: To the left, a sequence diagram shows a simplified GC algorithm where parts of the logic are moved into protected methods. This implementation can be refined through subclassing. To the right, the logic of one method was moved into the garbage strategy, a second method is wrapped.

7.2. Heap and Garbage Collector

The heap is responsible for managing objects in multiple memory regions, such as the boot image, the immortal heap and the object space. For the last, a garbage collector cleans up unreachable objects. A reusable heap implementation should allow adding new object spaces for parallel environments or has to be flexible enough to support objects of multiple layouts, as long as they fulfill the constraints for an integrated environment.

7.2.1. Garbage Strategy

A reusable heap implementation would have complex algorithms, such as the garbage collector, broken into parts of overridable methods, as illustrated on the left hand side of figure 7.3. If the variation points are chosen carefully, this allows for an easy customization of the heap via sub-classing.

However, even for separated object spaces, it is not necessary to have two heap instances. All that is needed is one additional memory region which is garbage collected, without a second boot image manager or immortal heap.

Instead, as shown on the right hand side of figure 7.3, the existing scheme instance can be extended by using the strategy pattern [13]. Selected parts of the GC algorithm are encapsulated in a "garbage strategy", which then implements these parts and/or invokes appropriate methods of the heap.

When a new memory region is added, only a customized garbage strategy has to be provided to enable garbage collection.

The garbage strategy interface is specific to the GC algorithm. For the semi-space garbage collector, the relevant variation points encapsulated by the strategy are

- the from- and to-space memory regions,
- the current to-space's allocation mark,
- the upper allocation bound and safety zone, and
- how, for a given object cell, references can be visited.

The advantage of this solution is that the heap still is unique, which simplifies the overall architecture with regard to the integrated environment approach. On the other hand, the garbage strategy does not allow using a different GC algorithm for the Java and the Guest VM. For this, it might be better to strengthen the notion of multiple heaps by moving the immortal heaps into a dedicated scheme.

7.2.2. Layout Responsibility

```
final SpecificLayout specificLayout = hub.specificLayout;
if (specificLayout.isTupleLayout()) {
   TupleReferenceMap.visitReferences(hub, origin, refUpdater);
   if (hub.isJLRReference) {
      SpecialReferenceManager.discoverSpecialReference(origin);
   }
   return cell.plus(hub.tupleSize);
}
if (specificLayout.isHybridLayout()) {
   TupleReferenceMap.visitReferences(hub, origin, refUpdater);
} else if (specificLayout.isReferenceArrayLayout()) {
   scanReferenceArray(origin);
}
return cell.plus(Layout.size(origin));
```

Listing 7.1: Scanning an object during garbage collection. In the original code, the GC holds strong assumptions how objects of certain specific layouts have to be scanned.

In the original implementation, shown in listing 7.1, the garbage collector holds strong assumptions about the layout implementation. It is expected that object references can only be found in tuples, in the tuple part of hybrids, and in reference arrays. Furthermore, even though the details of the layout are still hidden, it already implements a very specific solution to how the references can be found, by directly invoking appropriate utility functions.

To be independent from the layout implementation, the garbage collector already uses a visitor that is invoked for each reference in an object cell. As listing 7.2 shows, the distinction between layout kinds can be hidden completely using polymorphism. This solution even reduces the number of necessary interface calls from up to four (`isTupleLayout`, `isHybridLayout`, `isReferenceArrayLayout` and one hidden in `Layout.size`) to two.

```
final SpecificLayout specificLayout = hub.specificLayout;
specificLayout.visitReferences(hub, origin, refUpdater);
return cell.plus(specificLayout.size(origin));
```

Listing 7.2: Object scanning deferred to layout. Here, the GC completely relies on the specific layout to scan an object.

It should be noted that this change is complementary to the proposal that a garbage strategy defines how object cells are visited. The code shown here requires the object's specific layout and expects it can be found via the hub. The garbage strategy might define different ways of obtaining the special layout, e.g., by reading the object's base header. Furthermore, this change is just a special case of the more generic variation point "Predefined Layout Kinds".

7.3. Layout

The layout manages how data is stored inside of an object. It should provide direct and efficient access and, at the same time, has to be generic enough to support new object types that can be found in dynamic object-oriented languages but are unknown to Java.

7.3.1. Predefined Layout Kinds

Maxine defines three kind of objects: tuples, hybrids and arrays. Only the last kind has multiple instances, one for each content type. These three kinds are defined explicitly in the enum `Layout.Category`, and implicitly in methods such as `isTupleLayout` (see table 5.1 and table 5.2 on page 59 for similar methods). The assumption that there are exactly these layout kinds is made in several other, non-layout parts of the VM, such as the garbage collector, as discussed in variation point "Layout Responsibility".

This assumption should be removed entirely, together with the enum and the respective identification methods. It prevents the introduction of new object kinds, such as the reference hybrid or the compiled method, and should be replaced with generic methods to access object cells efficiently.

This requires further analysis of code that relies on the layout, to show which impact this assumption has on the way objects are accessed. The following variation point shows how the layout interface was extended to remove the assumption from the semi-space garbage collector.

7.3.2. Reference Visitor

Previously, the assumption of the three layout kinds was helpful because it allowed to define, for a given task such as visiting all references in an object cell, specially optimized implementations for each possible case. It is not desirable to replace these implementations with a generic, inefficient solution. Instead, the `SpecificLayout` interface was extended to directly provide a method for this high-level task. For the existing OHM layout kinds tuple, hybrid and array, the implementation was simply copied from the original source (see listing 7.1).

In SqueakMaxine, this change allowed correct garbage collection for new layout kinds. As an example, listing 7.3 shows the specialized visiting algorithm for compiled methods. The location of references can be determined neither from the hub (as for tuples and hybrids) nor from one of the header fields (as for arrays), but depends on the object's first content word which stores the method header.

7.3.3. General Layout Responsibility

At the time of writing, the general layout serves two purposes: On the one hand, it is used as a base class for the layout hierarchy, implementing methods that are similar for all specific layout kinds (see figure 3.7 on page 29); on the other hand, it can be used to select the specific layout for a given object.

While it is useful to have a base class in the layout hierarchy to avoid code duplication, the purpose of the general layout should be redefined with regard to objects of other layout hierarchies using the same heap. It should

```
public void visitReferences(Hub hub, Pointer origin,
                            PointerIndexVisitor visitor) {
  // visit literals
  final int start = HEADER_WORDS + EXTRA_HEADER_WORDS;
  final int end = start + getNumberOfReferences(origin);
  for (int i = start; i < end; i++) {
     visitor.visit(origin, i);
  }
}

public int getNumberOfReferences(Pointer origin) {
  return (origin.getWord(0, METHOD_HEADER_INDEX).toInt() >> 10) & 0xff;
}
```

Listing 7.3: Visiting references of a compiled methods. Especially for complex objects such as compiled methods, the specific layout knows best how to scan for references.

be independent even of the chosen Java layout, and instead be compatible with every layout implementation that conforms the rules for layouts in an integrated environment. For instance, the default general layout would define that a hub header field is located at an object's origin, and that the hub references a specific layout.

For additional heaps of parallel environments, another general layout that fits the otherwise incompatible new layouts can be implemented.

7.3.4. Additional Header Fields

Header fields are used to store data that should not be accessible as freely as fields are, or is otherwise not considered to be part of the objects content. For Maxine, this is the hub, the miscellaneous bits, and for arrays and hybrids the length. For other languages, additional header fields might be required, for instance, the associated class actor for Squeak class objects.

Adding header fields after the object's origin can cause problems with field offsets, as discussed in section 3.3.3, and adding header fields to all objects is wasteful if they are used only occasionally.

A possible solution would be to add new header fields before the origin, which, when using the OHM layout as a base, would lead to a mix of HOM and OHM layout for objects with additional headers. For instance, it could be defined that the three lowest bits of the first header field indicate up to seven additional words before the origin. As the value of the hub header always is a 64-bit word aligned reference, thus indicating zero by default, this allows for an easy cell-to-origin conversion. This logic would have to be included in the general layout (see previous variation point) and would not affect Java objects in any other way.

To convert origin to cell pointers, the specific layout can be used, which can be obtained via the hub. However, as the Squeak layout shows, it might be desirable to have a different number of header fields even for objects of the same specific layout. In this case, the lowest hub-header bits can be used to encode that number. Here, it would only be possible to indicate up to six additional header words, as one bit configuration is already reserved for forward references. Furthermore, every time the hub is retrieved from the object header, the lowest bits have to be masked, which creates a small runtime overhead.

7.3.5. Logic Duplication with Compiler

As described in section 5.2, the fact the most of the layout could not be implemented in a meaningful way was not an actual problem. The parts of the VM that used the layout had to be adapted anyway.

One part, however, that should have depended on the layout did not: the compiler. With the old `CPS` compiler, the layout had to define code snippets to describe actions such as hub or field accesses to the compiler. In the current `C1X` implementation, however, the entire layout logic is duplicated in assembler definition, which is why the HOM layout could not be used in the first place. With the new Graal compiler, however, it should be possible to return to a snippet based approach that removes layout logic from the compiler's internals.

7.3.6. Actor Hierarchy

Although not part of the layout scheme, actors are tightly coupled with the object model and the layout implementation. For instance, each layout kind has its respective class actor subclass which contains specialized initialization code. To introduce new layout kinds, these class actor implementations have to be sub-classed.

For SqueakMaxine, it was necessary to add new constructors with additional parameters. In Squeak, for instance, array classes can be named freely, so the array class actor needed a new constructor with a name parameter.

The actors have to be analyzed for assumptions held in the code, as some of them might not be necessary and are just Java-specific. It also seemed that some methods were declared final just for performance reasons.

Another important requirement is the possibility to change classes at runtime, including the addition and removal of fields and methods; and even changing the inheritance order. This is not just done with remove the final-modifier from the respective fields holding field or method actor arrays.

Changing the fields requires resizing all instances of this class, including its subclasses. Adding or removing methods might require the hub to be replaced, when the size of the vTable has to be changed. Both can be done with a modified GC operation; here, maybe the garbage strategy can be used to introduce the appropriate changes.

7.4. Other Subsystems

Even though not relevant for implementing the object model, other parts of Maxine were affected by the changes as well.

7.4.1. Reference Scheme

In many dynamic object-oriented languages *everything is an object*, which means they face the same problems as Squeak representing simple values, such as integers, and can benefit from tagged references. However, there are only three bits available for tagging, and a language implementation might want to represent instances of more than one class as tagged references. This can lead to problems when mixing multiple language environments. Furthermore, tagging leads to a constant overhead every time the hub is accessed. Scala and JRuby have shown that a language implementation that uses only boxed integers, i.e., every integer object has a memory cell, can be reasonably fast.

Thus, the usefulness and performance impact of tagged references has to be subject of further evaluation. Nevertheless, we have shown that implementing tagged references is possible with minimal effort.

7.4.2. Inspector

The inspector very much relies on the fact that Maxine is a Java-only VM. To obtain the layout information for a Java class, it reads the class-file and recreates the actors. This obviously does not work for Squeak classes that are loaded from an image file. To get the inspector to correctly visualize Squeak objects, it was necessary to generate class files that contain the respective field and method definitions.

For Squeak objects, this could be done because the object models of Squeak and Java are rather similar. For other languages, however, it might not be possible to express the classes as Java class-files. Here, a more flexible approach is needed to directly copy the actor data from the inspected VM.

7.4.3. Run Scheme versus Startup Scheme

Before the VM can start executing application code, the run scheme initializes VM internals, which also includes Java-specific API classes in the JDK. When all VM features are up and running, it finds and invokes the application's entry point.

For Java, this was loading the main class and invoking its main method. In SqueakMaxine, at this point the image is loaded.

As Maxine always has to be a fully working Java VM, the Java initialization code has to be copied for each run scheme of every new language. This can be avoided by introducing a new start-up scheme, which performs the mandatory initialization and having a lighter run scheme, which only invokes the language-specific entry point.

8. Maxine Variation Points: An Evaluation

One purpose of this work and of the implementation of SqueakMaxine is the identification of missing variation points. These have been determined as evaluation of the application of the principles laid out in chapter 4 to Squeak on Maxine. Hence, in the following, for each application, the status quo in Maxine is explained, the findings of the application are discussed, and the necessary variation points are presented, if any.

8.1. Instruction Set Abstractions

Maxine provides ways to handle Java bytecode, constituting its instruction set. However, there is no *general* abstraction for bytecodes on its own. Even more, there are at least three instances of Java bytecode handling interface in the current Maxine.

THE CRI The compiler–runtime interface (CRI) is a component of Maxine that is meant to facilitate the interaction between Maxine and the standard JVM. As such, it plays an important role for the upcoming Graal compiler, that runs on both Maxine and the standard JVM. The CRI is also used by the C1X compiler. One of the key parts is its Java bytecode interface; the CRI has a list of all Java bytecodes as Java constants, as well as a streaming interface to iterate over a typical Java bytecode method, with additional functionality to decode bytecode arguments. The C1X and T1X compilers uses the bytecode interface of the CRI directly. The CRI is completely Java-specific.

TEMPLATE TAGS The tags used in T1X templates roughly match the Java bytecode set, with exceptions of type-specific tags or bytecodes not implemented by templates. In the mapping between bytecodes and templates, the CRI bytecode constants are used to identify bytecodes and the CRI streaming interface is used to obtain a bytecode's respective argument in a method.

The `T1X` template tags built upon the CRI bytecode constants and are hence Java-specific.

`T1X` REFERENCE MAP GENERATION As said in section 6.4.1, reference maps for `T1X` compiled methods are generated from a Java method's bytecode. This is done by *interpreting* the method's list of bytecode. The bytecode values are checked and the possible stack depth of the native method is determined from the kind of the bytecode. This is done entirely independent of the CRI or the `T1X` template tags. The `ReferenceMapInterpreter` has a completely decoupled logic from the other two. The generation of reference maps is Java specific.

Working with this interfaces has disadvantages: Not counting the implementation of the actual bytecode semantics in either `T1X` or `C1X`, adding a bytecode requires changes to at least three distinct places: the CRI list of bytecodes, the option handling for this bytecode in the streaming interface of the CRI, the reference map interpreter and the `T1X` template tags.

Moreover, the Java bytecode abstraction in Maxine is not meant to be extensible to any extend. Each location where bytecode handling is done deliberately is implemented in a way it appears "sealed"; subclassing, general interfaces, or other means of extension are ruled out on the basis that, regarding Java, the bytecode set has singleton semantics, i.e., there is only one Java bytecode set, and that this set seldom changes. For a pure Java VM, this assumptions are reasonable and allow for common optimizations of this often used code.

SqueakMaxine Findings As Squeak has a comparable, but in details different bytecode set to Java, Maxine's way of handling Java bytecode is used as a blueprint for Squeak bytecodes in Maxine. Due to the design of the Java bytecode abstraction in Maxine, re-use in an object-oriented sense is not possible, thus, the Java bytecode implementation is rather copied verbatim and adapted to Squeak. This approach suffers from the same problems and benefits from the advantages as the Java bytecode implementation in Maxine: four dispersed locations of source code dealing with bytecode as well as a non-extensible, but easily optimizable implementation.

Instruction Set Abstraction A general abstraction for IR instruction sets, i.e., bytecodes in most cases, is necessary within Maxine. At least Smalltalk, Lua, and Python are commonly compiled to bytecodes and brining such dynamic object-oriented languages to Maxine requires support for their instruction sets. While listing all bytecodes of a languages instruction set is inevitable in most cases, a *common interface and method of specifying bytecodes* should be devised. This should include a streaming interface akin to the `BytecodeStream` found in the CRI for Java. Such a streaming interface requires information about bytecode arguments. To keep the number of places where bytecodes are specified small, such arguments should be specified in proximity to their bytecodes. Furthermore, if the instruction set is stack-based, a bytecode's effect on the stack could be recorded in proximity to a bytecode's arguments. That way, the number of places to change when a bytecode is changed, added, or removed, would decrease from about four to one or two for Java, depending on the compiler.

Given such an interface or abstract component, an instruction set for a dynamic object-oriented language could be specified by just listing the languages bytecodes with adjacent specification of their arguments and stack impact.

8.2. The Template-based Just-In-Time Compiler

The major part of an execution model implementation is the execution engine, which for Maxine is any of its JITs. For SqueakMaxine, this is provided by a JIT that is based on Maxine's template-based non-optimizing `T1X` compiler. This JIT, `S1X`, is able to compile Squeak bytecodes to native machine code just like `T1X` for Java does.

Ideally, `S1X` should not re-implement necessary functionality that is already present in `T1X`. However, `S1X` currently is an altered verbatim copy of `T1X` for two main reasons. First, `T1X` is not language-agnostic by design. The template tag system of `T1X` is directly based on the CRI implementation of Java bytecode and quite strongly coupled to it. Second, while `T1X` is already factorized, the factorization axis is not the language implemented but the target platform, i.e., for the `T1X` compilation `T1XCompilation` a target specific `AMD64T1XCompilation` exists (cf. figure 8.1). This approach does not allow the common practice of specialization by subclassing in order to specialize the language implemented. Hence, the `T1X` infrastructure is duplicated for `S1X`. During the implementation of `S1X`, it became apparent that `T1X`, and hence `S1X`, has three layers of implementation; the *compilation*, the *templates*, and the *runtime*. Refer to table 8.1 for a high-level comparison of these layers.

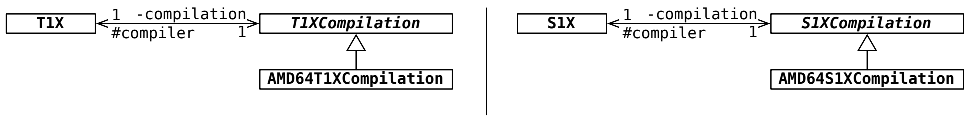


Figure 8.1.: Factorization of the template-based JIT compilers by target platform; left for T1X, right for S1X. Key: UML

Table 8.1.: Possible locations of bytecode functionality implementations.

	Compilation	Template code	Runtime
Class where implemented	`(AMD64)S1XCompilation`	`SqueakTemplateSource`	`S1XRuntime`
Example from listing 6.2	`do_prim_equivalent()`	`primitiveEquivalent()`	`trueReference()`
Functionality	stack manipulation	arithmetics, comparison	allocation
Limitations	very low level, needs per-architecture subclass	jumps, stack manipulation not expressible, may not allocate or call stubs	inlining restrictions

In any case, an implementation for each bytecode in the compilation is necessary, as the compilation has access to an assembler, which is beneficial certain cases like jumping. For most bytecodes, the compilation implementation, however, just uses the implementation given by the templates. In some cases, e.g., the creation of new objects, the templates have to call the runtime, as only there allocation is possible. This partitioning is reasonable, yet, figuring out what functionality to implement at which place can result in a trial-and-error approach. At least documentation of the capabilities of each layer is necessary.

Generic Template-based JIT To overcome the platform-preferring factorization in T1X, and to cope with different languages, a *generic template-based JIT* can be useful. It should provide the generally necessary infrastructure for writing JITs template-based, e.g., the recording and application of templates or the incrementing and decrementing of the stack pointer. Furthermore, a facility is necessary that provides a platform-independent means to express jumps, conditional or unconditional. Possibly, this could be derived from the existing Java and Squeak JIT implementations.

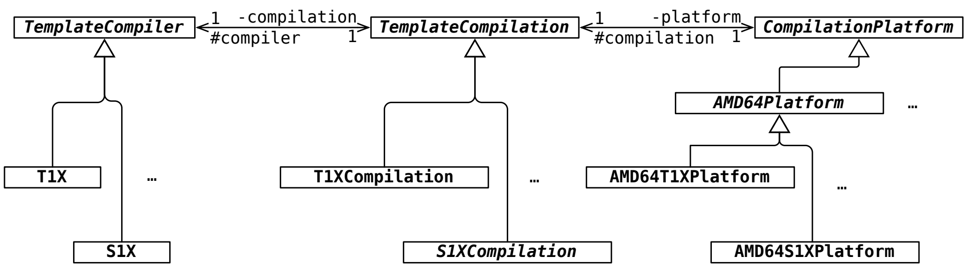


Figure 8.2.: Template compiler proposal. The factorization of the compiler is per-language. The per-platform code is a strategy, a possible approach to the two-dimensional nature of requirements to language and platform in T1X. Key: UML

From the implementation point of view, such an generalized template-based JIT should be factored by both language and platform. However, this is difficult, given that Java as the host language does not support multiple inheritance. Hence, the factorization should be dominated by the language implemented with the respective platform specific methods provided by a strategy. An example for such an architecture is given in figure 8.2. Additionally, providing templates the way T1X currently does is insufficient for multiple languages due to the restrictions of the Java annotations. However, SqueakMaxine already provides a means to circumvent these and to specify a *per-language template tag* that is able to maintain different sets of template tags within the compilation framework. This is already used for Squeak, as explained in section 6.3.2.

8.3. Re-usability of Maxine's Compilation Infrastructure

Although the compilation of dynamic object-oriented languages to machine code is different in detail, certain aspects of compilation are similar across languages. This includes the need to load and store temporary variables, write to or read from objects, or look up methods, among others. To ease the implementation process of JITs, those often-used compiler components should be available within Maxine. Actually, there are several of such re-usable components ready to use.

SNIPPETS Maxine snippets encapsulate often used functionality that is to be inlined into the code using it. Such functionality includes allocation of new objects with automatic linking to the Maxine internal actors, obtaining native code entry points from a given `TargetMethod`, or resolving classes, among others, with only the last being Java-specific.

ASSEMBLERS Maxine currently offers one programmatic native code assembler for AMD64 platforms. Furthermore, more abstract concepts exist as well, e.g., the `T1XCompilation` provides an interface to assign registers or peek from or poke to the stack, which could be re-used, given there is a generalized template compiler (cf. section 8.2).

RUNTIME INTERFACE The runtime interface of Maxine consist of functionality that is not compiled into native code methods by JITs at runtime, but functionality that is called from those code methods. An example is the `T1XRuntime` that encapsulates, e.g., object creation. Note that it makes us of snippets but itself provides a higher layer of abstraction. Also, a Java-specific compiler–runtime interface (CRI) exists that can partially be reused.

STACK WALKERS The Maxine stack walkers provide means to inspect the current execution and abstracts from the raw handling of frame pointers and stack pointers (cf. section 6.6.1)

MAXINE INTERMEDIATE REPRESENTATIONS Maxine has a lower-level intermediate representation (LIR) that can be used by optimizing compilers prior to generating actual machine code. Both the compilers and the LIR implementation can make use of XIR, that aims to be a compiler and runtime independent IR. It is possible to re-use either IR to interact with Maxine or re-use parts of other Maxine JITs.

OBJECT MODEL INTERACTION It is possible to interact with the object model implementation uniformly via the `Layout` utility class or its convenience wrappers like `ObjectAccess`. Directly modifying objects via pointers or references is not necessary if these interaction components are re-used.

COMPILER SELECTION Deciding which method to compile with which compiler is done in the compilation broker, which easily can be replaced by implementations aware of different languages (cf. section 6.3).

SqueakMaxine Findings We re-use snippets, an assembler, the runtime interface, stack walkers, and the object model interaction. Also, the variability of the compiler selection is used (cf. section 6.3.2). There is no new variation point necessary within Maxine's compilation infrastructure, as there are plenty of helper components and for the compiler selection, variation is already possible with the existing compilation broker. However, the broker should be extended to allow an arbitrary number of compiler fallbacks.

8.4. Implementing Built-in Behavior

While the notions of IR are quite similar among Java and dynamic object-oriented languages, their built-in behavior differs, even among each other. For example, Java's built-in behavior is based on its FFI, Python's built-in behavior is integrated in its module concept, and Smalltalk has a notion of methods tagged as primitives that are to be provided by the VM. These notions only have in common that, despite they all originate from a normal method call or message send, respectively, code is executed that does not exist in the languages common representation.

As a consequence of this divergence in notion, it is barely possible to provide an explicit but language-agnostic support for the built-in behavior of dynamic object-oriented languages. The road taken for SqueakMaxine is more an optimization, which is possible as a result of primitives being able to play the role of an "bytecode before the actual bytecode", than a general way for implementing built-in behavior.

Advanced Substitution The only concept Maxine could support regardless of the language implemented is an extended substitution mechanism. This comprises facilities that allow to match on methods of the language implemented and, subsequently, to replace the actual method to be executed by a custom one. This already exists in Maxine for Java. However, it is only possible to match the names of Java classes and methods by means of dedicated annotation. It should, however, be possible to specify the methods to match and replace in the way that is common to the language implemented.

Moreover, the code that actually carries out the substitution is integrated deeply with the Java compilers and, hence would have to be rewritten for every language that wants to use Maxine's substitution. Maxine should rather provide a more general substitution facility with means to query for methods that should be replaced and storage for the actual replacement to use. Several compilers then would be able to reuse such an infrastructure for replacing methods.

8.5. Stack Manipulation

The current Maxine implementation that provides access to the call stack information is rather difficult to understand and insufficient for dynamic object-oriented languages.

The former is because the lack of actual object-oriented abstraction for stack frames in Maxine, stack walker aside. There is no point where a stack frame is explicitly created. The mere fact that a *frame pointer* is altered constitutes the creation of a stack frame; its fields are just offsets, even at the higher level parts of Maxine. While there is a class `StackFrame`, it is only used when Maxine runs on top of a JVM in its hosted mode for, e. g., debugging by the inspector. Moreover, instances of this class do only represent the already existing stack; creating an instance of such object will not result in a new stack frame on the call stack.

The latter is caused by the circumstance that most dynamic object-oriented languages support a notion of anonymous units of execution. Common names for that include *closures* (which is used here), λ-expressions, or anonymous functions. A closure is able to access state from its enclosing context, i. e., the method defining the closure. Hence, a closure accessing a temporary variable in its enclosing context has to have access to the stack frame of that method during execution, as there is no other way to change the temporary variable. Furthermore, dynamic object-oriented languages like Smalltalk support the notion of *non-local return*, i. e., a closure is able to not only return from its own execution but also to return from the method it is defined in, regardless of how many different invocations are between them. Consequently, an implementation of closures with non-local returns has to have access to the stack frame of its defining method to obtain its return address to use it as its own return address.

Currently, it is possible to express the retrieval of the return address and the retrieval of temporary variable in a closure's enclosing context by means of a stack walker and an extension to the frame layout (cf. section 6.6.2). However, this is an read-only operation that is insufficient for the temporary variables, as for most dynamic object-oriented languages, especially for Smalltalk, it is expected that closures can write to its enclosing context's temporary variables.

Manual Stack Management Maxine needs to provide a way of manual stack management, i.e., means to allocate and modify the stack in a manner consistent with the rest of Maxine with respect to modularization and object-oriented abstraction. That way, the kind of stack access necessary for dynamic object-oriented languages that support closures can be provided. This may imply an advanced handling of the stack, e.g., by providing mechanisms similar to stack pages which are used in VMs like the CogVM [29]. Moreover, an explicit interaction with the stack could make garbage collection of running more approachable and probably even absorb the functionality of the current reference map interpreters.

An advanced, manual stack handling might have other beneficial uses. For example, it can be used to provide lightweight threading approaches like green threads in addition to the operating-system-level threads that are already present in Maxine. This can be useful for implementing functional languages like Erlang.

8.6. Applicability to Other Dynamic Languages

Implementing Squeak poses a first step in the process of bringing dynamic object-oriented languages to Maxine. The findings of SqueakMaxine suggest that the presented way of bringing a dynamic object-oriented language to Maxine is applicable to other dynamic object-oriented languages, with respect to their execution models. We think that the same restrictions that apply to implementing Squeak in Maxine also apply to other dynamic object-oriented languages and, hence, the variation points stated are in their case necessary, as well. Considerations for class-based and prototype-based languages are to follow.

8.6.1. Class-based Languages

Class-based dynamic object-oriented languages such as Python, Ruby, or Lua should be able to implement basically the same way as Squeak is; the class-notion of Maxine fits well and the main difference in the dynamic object-oriented languages' execution models is the respective lookup and the built-in behavior, which needs individual treatment in any case. However, certain language-specific notions in execution differ and might pose a challenge to Maxine.

Python In contrast to both Java and Squeak, Python supports multiple inheritance. Maxine does not anticipate this and probably needs dedicated support for such, essentially, lookup variants. Another, more challenging feature of execution in Python is the notion of *bound methods*: a method can be bound to a single receiver, effectively creating a new method that always operates on the receiver bound. In consequence, multiple instances of the very same method bound to different receivers can exist. This results in the need for some kind of code management to handle such instances. Python λ-expressions and nested functions have similar requirements as Squeak closures, but Python's notion of generators needs a more sophisticated approach of suspending and resuming the execution of methods. We expect that an even more intense interaction with and manipulation of the call stack is necessary to support generators than it is for Squeak contexts.

Ruby While Ruby does not provide multiple inheritance like Python, it does have a notion of sideways composition, called *mixins*. Such mixins can extend a class's functionality dynamically and, thus, might complicate the lookup and the generation of machine code, the latter, as the inclusion of a mixin can result in methods being present in the current class that were formerly take from a superclass. This at least calls for the possibility of deoptimization, which is possible in Maxine. Another peculiarity of Ruby is the notion of per-object meta-classes, commonly called *eigenclasses*, which can exist for every object and, subsequently can change the behavior of objects without changes to its nominal class. This can make an efficient storage and lookup of behavior, as done for Maxine by hubs for classes, challenging.

8.6.2. Prototype-based Languages

In contrast to class-based languages, prototype-based languages such as Self, JavaScript, or Io typically do not organize their behavior in classes. The distinction between an object's state and its behavior blurs; *slots* can contain either data or behavior at will. The class concept implemented in Maxine, that is exposed to its guest languages does not facilitate the non-class-based approach of organizing behavior. Moreover, while JavaScript only allows one direct parent object per object to search functionality for, which resembles Java's single inheritance, Self allows an arbitrary number of direct parents per object, effectively providing a kind of multiple inheritance.

An efficient approach to implement the per-object behavior of prototype-based languages has been proposed with the Self VM [7], grouping object with identical behavior in *object families*. Consequently, it is possible to have optimized data structures for these objects and to group behavior accordingly. This is not dissimilar to the current Maxine implementation with the notable difference that the Self object families are never visible in the guest language while the Maxine classes are. When implementing an object family approach in Maxine, a means to store such *family* information "out of sight" of the guest language is necessary, effectively the same requirement as explained in section 4.2.

In comparison with other dynamic object-oriented languages, prototype-based languages pose the biggest challenge regarding an implementation in Maxine. The mismatch between the class-based statically-typed approach of Maxine and the prototype-based dynamically-typed languages becomes apparent when considering possible implementations of certain language aspects, e. g., the vastly different, even object-local lookup; the object model differs even more.

9. Related Work

This chapter gives an overview of other work that is related to either Maxine itself or SqueakMaxine or. First, metacircular VMs and VMs written in programming languages with higher-level abstractions are presented, as those are most closely related to SqueakMaxine in its role as being implemented in a higher-level programming language within a metacircular VM. Next, VMs are presented that focus on VM configuration or generation, or on providing VM frameworks. These are in so fare related to Maxine as it aims to become a VM framework. Then, Squeak VMs and Java VM are presented, as these are connected to the original nature of either Squeak or Maxine. Finally, other dynamic object-oriented languages are presented that are present on JVMs, as they are, like SqueakMaxine, implementations of dynamic languages in Java.

9.1. Virtual Machines in Higher-level Languages and Metacircular Virtual Machines

Jikes, formerly Jalapeño [2], is quite similar to Maxine as it is a Java VM written in Java. However it does not use the standard JDK but an alternate runtime environment and, more importantly, does neither focus on configurability nor the inclusion of additional languages as done with SqueakMaxine.

The previously mentioned Squeak interpreter VM [21] is a VM that is written in itself in the sense that a subset of Smalltalk, called Slang, is translated to C and then compiled to the Squeak VM. It is allegedly the first metacircular virtual machine.

The PyPy [38] Python VM is written in RPython, a restricted subset of Python. The premises of both SqueakMaxine and PyPy are similar, however, PyPy more tries to be a tool chain for VM development with fixed combinable VM components, while Maxine focuses on configurability, with one configuration entity being the language, i. e., SqueakMaxine. One notable difference is that while for PyPy, the underlying language has to be restricted, for Maxine, the underlying language has been extended.

Other metacircular VMs include Klein [50], a Self VM written in Self, and the Rubinius[1] Ruby implementation.

9.2. Virtual Machine Product Lines, Platforms, and Frameworks

As pointed out, PyPy [38] aims to provide a tool chain for VM development; the implementations of different languages such as Squeak, Prolog, R, and Python allow a view on PyPy as a VM framework.

The Parrot VM[2], originally a Perl VM, aims to serve as a multi-language VM, however, in contrast to Maxine, it pursues the ideas of a common bytecode set for all languages that shall run on Parrot. Hence, dynamic object-oriented languages would run *on top* of Parrot while they could be run *inside* Maxine, as SqueakMaxine does.

LLVM [24] is a framework/VM hybrid that aims to provide some abstraction form the underlying machine while retaining as much control as possible over the details of generated code, object layout, an similar. It does not aim to provide object-oriented abstractions, however.

CSOM/PL [17] is a configurable VM product line, that allows the compile-time selection of features to be present in the resulting VM. It employs a specialized virtual machine architecture description language (VMADL) to describe the VM to be generated, unlike Maxine, which for this task uses a combination of Python scripts and Java classes with special meaning for the VM composition

9.3. Java Virtual Machines

The Oracle JVM with its HotSpot™ compiler is the reference implementation of the JVMs. It is written in C++, in this way differing from Maxine; as part of the *Da Vinci* project, the standard JVM is to become a multilingual VM, facilitating the implementation of dynamic object-oriented languages on top of

[1] `http://rubini.us`

[2] `http://www.parrot.org/`

the JVM by, e.g., providing special bytecodes for the method lookup in dynamic object-oriented languages. The aim of supporting multiple languages matches Maxine's but is carried out on a different level.

The Jikes VM[3], as pointed out, is a JVM written in Java. It pursues similar techniques as Maxine, with the main deviances being the supported platforms and the JRE used.

A quite different JVM is the Dalvik virtual machine. It supports the execution of Java programs after a transformation of the (stack-based) Java bytecode to Dalvik's own register-based bytecode. In that respect, Java is to Dalvik as Scala is to the JVM.

9.4. Squeak Virtual Machines

The root for all Squeak implementations is the original Squeak VM [21], that aims to provide an open Smalltalk-80 system. Initially, the original VM did only provide an interpreter-based execution, hence it is sometimes called interpreter VM. Although projects like Exupery[4] aim to provide a JIT, this VM primarily is used with its interpreter. Also, it serves as the reference implementation for other Squeak VMs and, hence, set the standard for SqueakMaxine with respect to feature-completeness.

Another well-known Squeak VM is Miranda's CogVM [28], providing a fully functional JIT for Squeak. However, unlike SqueakMaxine, Cog's primary concern is performance.

The RoarVM[5] [34, 49] is a quite recent implementation of a Squeak VM with a focus on exploiting manycore hardware with Squeak.

Potato[6] and its ancestor, JSqueak[7], are implementations of Squeak in Java, resulting in a dual stack VM. Potato is similar to SqueakMaxine with regard to language both are implemented in. In fact, they even share some minor design decisions, mostly regarding the loading of Squeak images.

SPy [5] is an implementation of Squeak in the PyPy framework. Although implemented in a subset of Python, RPython, it shares a similar purpose with SqueakMaxine: both were implemented to test the variability of their

[3]`http://jikes.sourceforge.net/`
[4]`http://wiki.squeak.org/squeak/Exupery`
[5]`http://soft.vub.ac.be/~smarr/renaissance/`
[6]`http://sourceforge.net/projects/potatovm/`
[7]`http://labs.oracle.com/projects/JSqueak/`

VM platforms. SPy is closer to feature-completeness, however, it is based on a rather old version of Squeak and does not handle newer, closure-based Squeak images. While only rudimentary, SqueakMaxine anticipates Squeak closures.

9.5. Dynamic Languages on Java Virtual Machines

Jython[8] is an implementation of Python in Java that runs on top of the standard JVM. It tries to leverage the benefits of a Java-based VM implementation and, in this respect, is similar to SqueakMaxine. As it runs on top of the JVM, a double stack VM scenario cannot be avoided. Jython supports the JIT compilation of Python source code to JVM bytecode.

JRuby[9] is a similar project as Jython, but for the Ruby language. However, JRuby tries to benefit from the rich standard library of the JDK, in this respect deviating from the default Ruby implementation. Its relation to SqueakMaxine is the same as Jython's to SqueakMaxine.

Scala[10] is a dynamic object-oriented language with optional typing that is designed to run on top of the JVM. It, hence, requires its source code to be compiled to Java bytecode in a different step than it is executed, which sets Scala apart from JRuby and Jython. Even more, compiled Scala applications are hardly distinguishable from ordinary Java programs. Scala differs from SqueakMaxine in type discipline, compile/run cycle, and most important, intent; Scala tries to leverage the Java bytecode for a dynamically typed language, SqueakMaxine tries to not use it at all.

[8]http://jython.org/

[9]http://jruby.org/

[10]http://www.scala-lang.org/

10. Summary and Future Work

This work presented an approach for bringing the object and execution models of dynamic object-oriented languages to the Maxine VM. We implemented Squeak / Smalltalk using this approach and, in this course, identified variation points. These variation points are necessary to facilitate the implementation of dynamic object-oriented languages in Maxine and support Maxine in becoming a VM framework, subsequently.

The approach for mapping execution models to Maxine includes the definition of execution models in this work and conceptual processes of how to implement individual parts of such an execution model in Maxine. To verify these processes, and to identify missing variation points in Maxine, they were carried out on an implementation of Squeak / Smalltalk in Maxine. We expect the identified variation points to improve the capabilities of Maxine to provide execution for arbitrary dynamic object-oriented languages.

Open Tasks for SqueakMaxine The implementation of Squeak in Maxine, SqueakMaxine, has served its purpose as feasibility study for dynamic object-oriented languages in Maxine. However, SqueakMaxine is yet to be completed to be regarded as an actual Squeak VM. Specifically, the closure support is still rudimentary and has to be fully implemented. Up until now, there is no support for the Squeak graphical user interface. Likewise, only a subset of the Squeak primitives is currently available in SqueakMaxine and awaits finalization. These should be mere implementation tasks once the required variation points are present in Maxine.

More elaborate tasks for SqueakMaxine include the support of traditional Squeak plugins, a means to extend Squeak by functionality not written in Smalltalk, as well as the Squeak FFI, a more recent approach to the same idea. These concepts commonly are provided by JNI in the Java world, however, they are mutually incompatible and, hence, can constitute an interesting follow-up question to this work.

Steps towards a Virtual Machine Framework Up until now, Maxine is a single language VM. Provided the identified variation points are added to Maxine, implementing dynamic object-oriented languages should be feasible. However, some concepts have not been tested with the SqueakMaxine implementation and remain for future consideration.

As already pointed out, the Graal compiler is intended to also support the implementation of dynamic object-oriented languages, but while it is important to have optimizing compilers for dynamic object-oriented languages in Maxine, the feasibility of Graal for that task has not been tested yet. Likewise, the only FFI variant present in Maxine is the Java native interface (JNI), which is powerful but not supported by all dynamic object-oriented languages; those typically com with their own FFI flavor. Probably a mapping is possible there, but this remains to be explored.

A next big step toward multiple languages in Maxine would be the possibility to add, remove, or exchange language modules for Maxine at runtime. Such a pluggability approach could lower the entry barrier and catalyze the implementation of a quite large number of dynamic object-oriented languages in Maxine. framework for dynamic object-oriented languages has been taken.

Systems have sub-systems
and sub-systems have sub systems
and so on ad infinitum—
which is why we're always starting over.
—Alan Perlis

Bibliography

[1] Norman I. Adams IV, David H. Bartley, Gary Brooks, R. Kent Dybvig, Daniel Paul Friedman, Robert Halstead, Chris Hanson, Christopher Thomas Haynes, Eugene Kohlbecker, Don Oxley, Kent M. Pitman, Guillermo Juan Rozas, Guy Lewis Steele Jr., Gerald Jay Sussman, Mitchell Wand, and Harold Abelson. "Revised5 report on the algorithmic language scheme." In: *SIGPLAN Not.* 33 (9 1998-09), pages 26–76. ISSN: 0362-1340. DOI: `10.1145/290229.290234`.

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. "Implementing Jalapeño in Java." In: *SIGPLAN Not.* 34.10 (1999-10), pages 314–324. ISSN: 0362-1340. DOI: `10.1145/320385.320418`.

[3] American National Standards Institute. *ANSI NCITS 319-1998 (R2007): Information Technology – Programming Languages – Smalltalk*. New York, NY, USA, 1998.

[4] David Bacon, Stephen Fink, and David Grove. "Space- and Time-Efficient Implementation of the Java Object Model." In: *ECOOP 2002*. Edited by Boris Magnusson. Volume 2374. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pages 13–27. DOI: `10.1007/3-540-47993-7-5`.

[5] Carl Friedrich 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." In: *Self-Sustaining Systems*. Volume 5146. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pages 123–139. DOI: `10.1007/978-3-540-89275-5_7`.

[6] Carl Friedrich Bolz and Armin Rigo. "How to not write virtual machines for dynamic languages." In: *3rd Workshop on Dynamic Languages and Applications (Dyla'07)*. Berlin, Germany, 2007.

[7] Craig Chambers, David Ungar, and Elgin Lee. "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes." In: *Conference proceedings on Object-oriented programming systems, languages and applications*. OOPSLA '89. New Orleans, Louisiana, United States: ACM, 1989, pages 49–70. ISBN: 0-89791-333-7. DOI: 10.1145/74877.74884.

[8] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. 3rd. SEI Series in Software Engineering. Boston, MA, USA: Addison-Wesley, 2001-08-30. ISBN: 978-0201703320.

[9] Melvin E. Conway. "Design of a separable transition-diagram compiler." In: *Commun. ACM* 6.7 (1963-07), pages 396–408. ISSN: 0001-0782. DOI: 10.1145/366663.366704.

[10] L. Peter Deutsch and Allan M. Schiffman. "Efficient implementation of the Smalltalk-80 system." In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '84. Salt Lake City, Utah, United States: ACM, New York, NY, USA, 1984, pages 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542.

[11] ECMA. *ECMA-262: ECMAScript Language Specification*. Third. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), 1999-12.

[12] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, 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. Washington, DC, USA: ACM, 2009, pages 81–90. ISBN: 978-1-60558-375-4. DOI: 10.1145/1508293.1508305.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[14] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Commonly called the "Blue Book". Boston, MA, USA: Addison-Wesley, 1983. ISBN: 0-201-11371-6.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. 2nd. Boston, MA, USA: Addison-Wesley, 2000. ISBN: 0-201-31008-2.

[16] Michael Haupt and Robert Hirschfeld. *Virtual Machines: 3. High-Level Language VMs, A Tour of SOM++*. Licensed under Creative Commons Attribution-Share Alike 3.0 Germany 🅭🅯🄎. Lecture. Hasso-Plattner-Institut, University of Potsdam. Potsdam, Germany, 2009.

[17] Michael Haupt, Stefan Marr, and Robert Hirschfeld. "CSOM/PL — A Virtual Machine Product Line." In: *Journal of Object Technology* 10 (2011), 12:1–30. ISSN: 1660-1769. DOI: 10.5381/jot.2011.10.1.a12.

[18] C. A. R. Hoare. "Monitors: an operating system structuring concept." In: *Commun. ACM* 17.10 (1974-10), pages 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161.

[19] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging optimized code with dynamic deoptimization." In: *SIGPLAN Not.* 27 (7 1992-07), pages 32–43. ISSN: 0362-1340. DOI: 10.1145/143103.143114.

[21] 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: *SIGPLAN Not.* 32 (10 1997-10), pages 318–326. ISSN: 0362-1340. DOI: 10.1145/263700.263754.

[22] Alan Kay and Stefan Ram. *Dr. Alan Kay on the Meaning of "Object-Oriented Programming"*. Email conversation between Alan Kay and Stefan Ram. 2003-07-23. URL: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en (visited on 2012-01-09).

[23] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. "Design of the Java HotSpot™ client compiler for Java 6." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5 (1 2008-05), 7:1–7:32. ISSN: 1544-3566. DOI: 10.1145/1369396.1370017.

[24] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pages 75–86. ISBN: 0-7695-2102-9. DOI: 10.1109/CGO.2004.1281665.

[25] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. 2nd. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0201432943.

[26] Bernd Mathiske. "The Maxine Virtual Machine and Inspector." In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA'08. Nashville, TN, USA: ACM, 2008, pages 739–740. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449838.

[27] Eliot Miranda. *Context Management in VisualWorks 5i*. Technical report. ParcPlace Division, CINCOM, Inc, 1999.

[28] Eliot Miranda. "The Cog Smalltalk Virtual Machine: writing a JIT in a high-level dynamic language." Talk given at the 5th workshop on Virtual Machines and Intermediate Languages VMIL'11. Unpublished. 2011-10-24.

[29] Eliot Miranda. *Under Cover Contexts and the Big Frame-Up*. 2009-01-14. URL: http://www.mirandabanda.org/cogblog/2009/01/14/under-cover-contexts-and-the-big-frame-up/ (visited on 2012-02-29).

[30] NCITS Technical Committee H7. *X3H7-93-007v12b: Object Model Features Matrix*. 1997-05-25. URL: http://www.objs.com/x3h7/fmindex.htm (visited on 2011-08-18).

[31] Oracle, Inc. *The Java HotSpot Performance Engine Architecture. Java HotSpot Client Compiler*. 2003-04. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html#client (visited on 2012-01-04).

[32] Oracle, Inc. *The Java HotSpot Performance Engine Architecture. Dynamic Deoptimization*. 2003-04. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html#dynamic (visited on 2012-01-04).

[33] Oracle, Inc. *The Java HotSpot Performance Engine Architecture*. 2003-04. URL: http://www.oracle.com/technetwork/java/whitepaper-135217.html (visited on 2012-01-04).

[34] J. Pallas and D. Ungar. "Multiprocessor Smalltalk: a case study of a multiprocessor-based programming environment." In: *SIGPLAN Not.* 23.7 (1988-06), pages 268–277. ISSN: 0362-1340. DOI: 10.1145/960116.54017.

[35] Python Software Foundation. *dis — Disassembler for Python bytecode*. 2011-12-13. URL: http://docs.python.org/py3k/library/dis.html (visited on 2011-12-13).

[36] Python Software Foundation. *Execution model*. 2011-12-13. URL: `http://docs.python.org/py3k/reference/executionmodel.html` (visited on 2011-12-13).

[37] Python Software Foundation. *Python Data model — Special method names*. 2011-12-13. URL: `http://docs.python.org/py3k/reference/datamodel.html#special-method-names` (visited on 2011-12-13).

[38] Armin Rigo and Samuele Pedroni. "PyPy's approach to virtual machine construction." In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pages 944–953. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176753.

[39] John R. Rose. "Bytecodes meet combinators: invokedynamic on the JVM." In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. VMIL '09. Orlando, Florida: ACM, 2009, 2:1–2:11. ISBN: 978-1-60558-874-2. DOI: 10.1145/1711506.1711508.

[40] Douglas Simon. "XIR definition." personal communication. 2012-01-05.

[41] Douglas Simon and Michael van de Vanter. *Maxine Schemes*. 2012. URL: `https://wikis.oracle.com/display/MaxineVM/Schemes` (visited on 2012-04-04).

[42] Douglas Simon and Michael van de Vanter. *Maxine VM*. 2011-11-09. URL: `http://wikis.oracle.com/pages/viewpage.action?pageId=31394883` (visited on 2011-12-05).

[43] Douglas Simon, Michael van de Vanter, and Ben Titzer. *C1 X - Maxine VM*. 2011-04-05. URL: `http://wikis.oracle.com/pages/viewpage.action?pageId=31394883` (visited on 2011-12-05).

[44] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[45] Alan Snyder. *An Abstract Object Model for Object-Oriented Systems*. Technical report. HP Laboratories, 1990-04.

[46] *Squeak Smalltalk*. 2012. URL: `http://squeak.org/Smalltalk/` (visited on 2012-04-04).

[47] Christopher Strachey and Christopher P. Wadsworth. "Continuations: A Mathematical Semantics for Handling Full Jumps." In: *Higher-Order and Symbolic Computation* 13 (2000), pages 135–152. DOI: 10.1023/A:1010026413531.

[48] Ben L. Titzer, Thomas Würthinger, Doug Simon, and Marcelo Cintra. "Improving compiler-runtime separation with XIR." In: *Proceedings of the 6th International Conference on Virtual Execution Environments, VEE 2010, Pittsburgh, Pennsylvania, USA, March 17-19, 2010*. Edited by Marc E. Fiuczynski, Emery D. Berger, and Andrew Warfield. VEE '10. New York, NY, USA: ACM, 2010, pages 39–50. ISBN: 978-1-60558-910-7. DOI: 10.1145/1735997.1736005.

[49] David Ungar and Sam S. Adams. "Hosting an object heap on many-core hardware: an exploration." In: *SIGPLAN Notices* 44.12 (2009-10), pages 99–110. ISSN: 0362-1340. DOI: 10.1145/1837513.1640149.

[50] David Ungar, Adam Spitz, and Alex Ausch. "Constructing a metacircular Virtual machine in an exploratory programming environment." In: *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pages 11–20. ISBN: 1-59593-193-7. DOI: 10.1145/1094855.1094865.

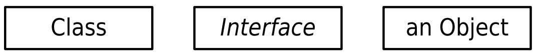
[51] Michael van de Vanter. *Schemes - Maxine VM*. 2011-10-28. URL: https://wikis.oracle.com/pages/viewpage.action?pageId=4161615 (visited on 2011-12-06).

[52] Mario Wolczko. "Semantics of Smalltalk-80." In: *ECOOP' 87 European Conference on Object-Oriented Programming*. Edited by Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman. Volume 276. Lecture Notes in Computer Science. London, UK: Springer Berlin / Heidelberg, 1987, pages 108–120. ISBN: 978-3-540-18353-2. DOI: 10.1007/3-540-47891-4_11.

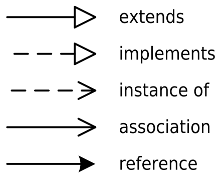
Appendix A.

Key to Figures in Chapters 3 and 5

Throughout this work, a variation of UML class and object diagrams is used to illustrate structural concepts. UML was chosen as a base for it is an expressive and well-known modeling language. At some points the notion is simplified to avoid unnecessary complexity, other points are extended to allow for more precision when describing the state of objects.

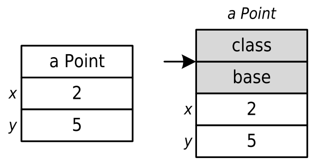


Classes and objects are shown as rectangles. Objects are usually labelled in lower case, if possible with a descriptive name. Classes are labelled in upper case, even if they are objects at the same time. Unless relevant, field and method definitions are omitted. Java interfaces are set in *italics* and can also be identified by *implements*-connectors.



To show relations between entities, the default arrow types are used for subclassing, implementing interfaces (Java only) and instantiating a class. An association between classes defines that instances of these classes may reference each other. Associations usually have a name and multiplicity information; arrow heads indicate whether they are uni- or bi-directional. For better distinction, actual references between objects are shown with a different arrow type.

Sometimes it is necessary to show details of an object's state. In these cases, one of two notations is used: the logical and the physical representation.



In the logical representation, fields are shown as additional boxes. They contain either a string representation of their value or are the origin of a reference connector. Information that is not logically stored in the object, but is shown for a better understanding of the values, is placed outside of the boxes. In the example shown above, they are the field names that are only known to the class, but not to the object itself.

The physical representation shows how the object is stored in memory. Here, each box represents one word. Header fields are highlighted gray and labelled with their name, regular fields are shown similar to the logical representation. A small arrow indicates the object's origin, the address that is used to reference the object. Again, complementary information may be shown outside of the cell.

Other diagram types or object connectors are explained as they are used.

Appendix B.

Extended Interfaces

Table B.1.: To allow for a modular and configurable architecture of SqueakMaxine, the existing scheme interfaces were extended with the methods listed here.

Method	Description
SqueakLayoutScheme	
`boolean isSqueakObject(Accessor obj)`	Determines whether the parameter references a Squeak object.
`Pointer nextObject(Pointer orig)`	For a given object, returns the next object on the heap.
`SqueakImageLoaderScheme getSqueakImageLoader()`	Returns the image loader, which populates the heap from a Squeak image file (see below).
`SqueakRuntime getSqueakRuntime()`	Returns the Squeak runtime, which provides reflection methods for Squeak objects (see below).
SqueakRuntime	
`Reference getSpecialObjectReference( SpecialObject so)`	Returns the requested special object; `enum SpecialObject` defines well-known object indices.

`Reference getInstanceVariable(Reference object, Index index)`
`void setInstanceVariable(Reference o, Index i, Reference v)`
	Accesses an object's field by index. As Squeak uses one-based indices, while Java by default uses zero-based indices, the meaning of an `int` parameter would be ambigous. Instances of `Index` explicitly provide a zero and a one-based value.

`Reference getInstanceVariable(Reference o, String n)`
`void setInstanceVariable(Reference o, String n, Reference v)`
	Accesses an object's field by name.

`Reference getClass(Reference object)`	Returns the class object for a given Squeak object.
`String toJavaString(Reference object)`	Returns a simple string representation of the object.

Table B.1.: To allow for a modular and configurable architecture of SqueakMaxine, the existing scheme interfaces were extended with the methods listed here. (continued)

Method	Description
	SqueakRuntime (continued)
`Reference getSmallIntegerReferene(long value)` `long getSmallIntegerValue(Reference smallInteger)`	Converts Java integers to and from tagged integer references.
`int getArrayLength(Reference array)`	Returns the length of a Squeak array.
`Reference getElement(Reference array, int index)` `void setElement(Reference array, int index, Reference value)`	Accesses values of an object array.
`byte getByte(Reference array, int index)` `void setByte(Reference array, int index, byte value)`	Accesses values of a byte array.
`Word getWord(Reference array, int index)` `void setWord(Reference array, int index, Word value)`	Accesses values of a word array.
`int getFirstByteCodeIndex(Reference compiledMethod)` `int getLastByteCodeIndex(Reference compiledMethod)`	Determines the bytecode part in a compiled method. Bytes can be accessed with `getByte`.
`int getNumberOfLiterals(SqueakReference compiledMethod)`	Determines the number of literal references in a compiled method. Literals can be accessed with `getElement`.
`int getNumberOfArguments(SqueakReference compiledMethod)` `int getNumberOfTemporaries(SqueakReference compiledMethod)`	Returns the number of parameters and temporal variables of a compiled method.
	SqueakImageLoaderScheme
`void loadImage(File image)throws IOException, ImageFormatException`	Loads an image file.

Table B.1.: To allow for a modular and configurable architecture of SqueakMaxine, the existing scheme interfaces were extended with the methods listed here. (continued)

Method	Description
	SqueakHeapScheme
`**boolean** supportsSqueakObjectDetection()`	Returns whether the heap is capable of distinguishing between Squeak and Java objects on its own. True for the dual semi-space heap of the Squeak layout approach, false for Maxine layout approach.
`**boolean** isSqueakObject(Accessor object)`	If Squeak object detection is supported, returns whether the argument references a Squeak object.
`Reference specialObjectsArray()`	Returns the special objects array.
`Pointer imageStart()` `Pointer imageEnd()`	Returns the start and end of the current to-space.
`SqueakReference firstObject()`	Returns the first Squeak object on the heap.
`ImageLoadingAllocator getImageLoadingAllocator()`	Returns the image loading allocator (see below), which is used by the image loader to populate the heap.

Table B.1.: To allow for a modular and configurable architecture of SqueakMaxine, the existing scheme interfaces were extended with the methods listed here. (continued)

Method	Description
	ImageLoadingAllocator
`Region placeImageIntoMemory(File imageFile)throws IOException`	Places the content of the image file in memory.
`void reserveMemory(Size requestedMemory)`	Reserves memory for allocating Squeak objects.
`Region getPrivateMemorySpace()`	Returns the reserved memory region. It can be filled with Squeak objects; objects created with the `new`-keyword will not be allocated here.
`void informLoadingDone()`	Informs the heap that loading is completed. For instance, the image file memory can be freed now.
`void setSpecialObjectsArray( Reference soa)`	Makes the special objects array known to the heap.
	SqueakReferenceScheme

`boolean isMarkedSqueak(Reference ref)`
`Reference markedSqueak(Reference ref)`
`Reference unmarkedSqueak( Reference ref)`
	Handles forward references explicitly for Squeak objects. Only used by the Squeak layout approach, which uses a different bit to indicate forward references. With the Maxine layout approach, Java and Squeak objects are handled the same way.

`boolean isTaggedInteger(Reference object)`
`Reference fromLong(long value)`
`long toLong(Reference object)`
	Implements how small integers are realized as tagged references.

Appendix C.

Squeak Essential Primitives

Table C.1.: Squeak primitives that are marked "Essential.". A total of 65 unique primitives constitute the built-in behavior of Squeak/Smalltalk; including arithmetics, object content access, instance creation, and invocation, among others. Some methods share the same primitive; those primitives' indices are only given once in the table.

Index	Class	Method	Category
1	`SmallInteger`	`+`	*arithmetic*
2	`SmallInteger`	`-`	*arithmetic*
3	`SmallInteger`	`<`	*comparing*
4	`SmallInteger`	`>`	*comparing*
7	`SmallInteger`	`=`	*comparing*
8	`SmallInteger`	`~=`	*comparing*
9	`SmallInteger`	`*`	*arithmetic*
12	`SmallInteger`	`//`	*arithmetic*
14	`SmallInteger`	`bitAnd:`	*bit manipulation*
15	`SmallInteger`	`bitOr:`	*bit manipulation*
16	`SmallInteger`	`bitXor:`	*bit manipulation*
17	`SmallInteger`	`bitShift:`	*bit manipulation*
38	`Float`	`basicAt:`	*accessing*
39	`Float`	`basicAt:put:`	*accessing*
40	`SmallInteger`	`asFloat`	*converting*
41	`Float`	`+`	*arithmetic*
42	`Float`	`-`	*arithmetic*
43	`Float`	`<`	*comparing*
44	`Float`	`>`	*comparing*
47	`Float`	`=`	*comparing*
49	`Float`	`*`	*arithmetic*
50	`Float`	`/`	*arithmetic*
51	`Float`	`truncated`	*truncation and round off*
60	`Object`	`basicAt:`	*accessing*
	`Object`	`at:`	*accessing*
	`LargePositiveInteger`	`digitAt:`	*system primitives*

Table C.1.: Squeak primitives that are marked "Essential." (continued)

Index	Class	Method	Category
	`FutureMaker`	`basicAt:`	*accessing*
61	`WideSymbol`	`pvtAt:put:`	*private*
	`Object`	`basicAt:put:`	*accessing*
	`Object`	`at:put:`	*accessing*
	`LargePositiveInteger`	`digitAt:put:`	*system primitives*
	`FutureMaker`	`basicAt:put:`	*accessing*
62	`Object`	`size`	*accessing*
	`Object`	`basicSize`	*accessing*
	`LargePositiveInteger`	`digitLength`	*system primitives*
	`FutureMaker`	`basicSize`	*accessing*
63	`ByteSymbol`	`at:`	*accessing*
	`ByteString`	`at:`	*accessing*
64	`ByteSymbol`	`pvtAt:put:`	*private*
	`ByteString`	`at:put:`	*accessing*
68	`CompiledMethod`	`objectAt:`	*literals*
69	`CompiledMethod`	`objectAt:put:`	*literals*
70	`Interval class`	`new`	*instance creation*
	`Behavior`	`basicNew`	*instance creation*
71	`Behavior`	`basicNew:`	*instance creation*
73	`ObjectOut`	`xxxInstVarAt:`	*basics*
	`Object`	`instVarAt:`	*system primitives*
	`FutureMaker`	`instVarAt:`	*accessing*
74	`ObjectOut`	`xxxInstVarAt:put:`	*basics*
	`Object`	`instVarAt:put:`	*system primitives*
	`FutureMaker`	`instVarAt:put:`	*accessing*
75	`ProtoObject`	`identityHash`	*comparing*
77	`Behavior`	`someInstance`	*accessing instances and variables*
78	`ProtoObject`	`nextInstance`	*system primitives*
79	`CompiledMethod class`	`newMethod:header:`	*instance creation*
82	`BlockContext`	`valueWithArguments:`	*evaluating*
85	`Semaphore`	`signal`	*communication*
86	`Semaphore`	`wait`	*communication*
87	`Process`	`primitiveResume`	*changing process state*
89	`Behavior`	`flushCache`	*private*
94	`EventSensor`	`primGetNextEvent:`	*private-I/O*
97	`SmalltalkImage`	`snapshotPrimitive`	*snapshot and quit*
100	`Object`	`perform:withArguments:inSupe`	*message handling*
101	`Cursor`	`beCursorWithMask:`	*primitives*
	`Cursor`	`beCursor`	*primitives*
102	`DisplayScreen`	`beDisplay`	*private*
110	`ProtoObject`	`==`	*comparing*
111	`ObjectOut`	`xxxClass`	*basics*
	`Object`	`class`	*class membership*

Table C.1.: Squeak primitives that are marked "Essential." (continued)

Index	Class	Method	Category
	`ImageSegmentRootStub`	`xxxClass`	*fetch from disk*
113	`SmalltalkImage`	`quitPrimitive`	*snapshot and quit*
114	`SmalltalkImage`	`exitToDebugger`	*snapshot and quit*
	↑ `Smalltalk`	`Squeak/Smalltalk only` ↓	
136	`Delay class`	`primSignal:atMilliseconds:`	*primitives*
137	`Time class`	`primSecondsClock`	*smalltalk-80*
201	`BlockClosure`	`value`	*evaluating*
202	`BlockClosure`	`value:`	*evaluating*
203	`BlockClosure`	`value:value:`	*evaluating*
204	`BlockClosure`	`value:value:value:`	*evaluating*
205	`BlockClosure`	`value:value:value:value:`	*evaluating*
206	`BlockClosure`	`valueWithArguments:`	*evaluating*
210	`MethodContext`	`tempAt:`	*accessing*
	`ContextPart`	`basicAt:`	*accessing*
	`ContextPart`	`at:`	*accessing*
211	`MethodContext`	`tempAt:put:`	*accessing*
	`ContextPart`	`basicAt:put:`	*accessing*
	`ContextPart`	`at:put:`	*accessing*
212	`ContextPart`	`size`	*accessing*
	`ContextPart`	`basicSize`	*accessing*
221	`BlockClosure`	`valueNoContextSwitch`	*evaluating*
222	`BlockClosure`	`valueNoContextSwitch:`	*evaluating*
(117)	`BitBlt`	`copyBitsAgain` [1]	*private*

[1]This primitive is actually not an indexed but a named primitive, yet marked as being essential. We suspect that this is a bug.

Appendix D.

SqueakMaxine Primitive Implementation Overview

Table D.1.: Squeak primitives and their implementation status in SqueakMaxine. The name column refers to the primitive name in the Squeak VM.

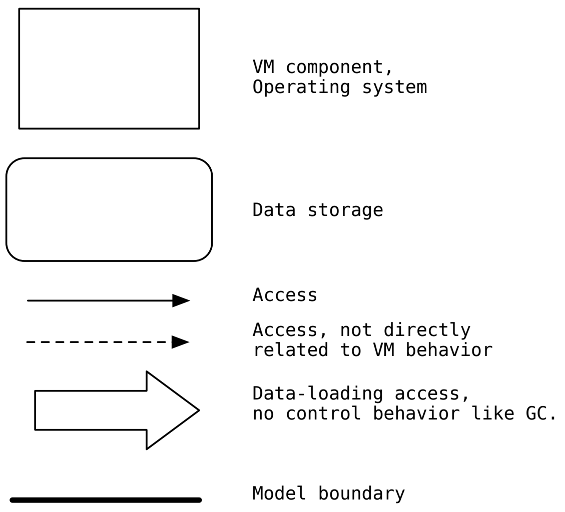
Index	Essential	Implemented	Name	*Remark*
1	✔	✔	Add	
2	✔	✔	Subtract	
3	✔	✔	LessThan	
4	✔	✔	GreaterThan	
5		✔	LessOrEqual	
6		✔	GreaterOrEqual	
7	✔	✔	Equal	
8	✔	✔	NotEqual	
9	✔	✔	Multiply	
10		✔	Divide	
12	✔	✔	Div	
14	✔	✘	BitAnd	
15	✔	✘	BitOr	
16	✔	✘	BitXor	
17	✔	✔	BitShift	
38	✔	✘	FloatAt	*not interpreter*
39	✔	✘	FloatAtPut	*not interpreter*
40	✔	✘	AsFloat	*not interpreter*
41	✔	✘	FloatAdd	*not interpreter*
42	✔	✘	FloatSubtract	*not interpreter*
43	✔	✘	FloatLessThan	*not interpreter*
44	✔	✘	FloatGreaterThan	*not interpreter*
47	✔	✘	FloatEqual	*not interpreter*
49	✔	✘	FloatNotEqual	*not interpreter*
50	✔	✘	FloatMultiply	*not interpreter*
51	✔	✘	FloatDivide	*not interpreter*
60	✔	✔	At	

Table D.1.: Squeak primitives and their implementation status in SqueakMaxine. (continued)

Index	Essential	Implemented	Name	*Remark*
61	✔	✘	AtPut	
62	✔	✘	Size	
63	✔	✘	StringAt	
64	✔	✘	StringAtPut	
68	✔	✘	ObjectAt	
69	✔	✘	ObjectAtPut	
70	✔	✔	New	
71	✔	✔	NewWithArg	
73	✔	✘	InstVarAt	
74	✔	✘	InstVarAtPut	
75	✔	✘	IdentityHash	
77	✔	✘	SomeInstance	
78	✔	✘	NextInstance	
79	✔	✘	NewMethod	
82	✔	✘	ValueWithArgs	*only for contexts*
85	✔	✘	Signal	
86	✔	✘	Wait	
87	✔	✘	Resume	
89	✔	✘	FlushCache	
94	✔	✘	GetNextEvent	
97	✔	✘	Snapshot	
100	✔	✘	PerformInSuperclass	
101	✔	✘	BeCursor	
102	✔	✘	BeDisplay	
110	✔	✔	Identical	
111	✔	✔	Class	
113	✔	✘	Quit	
114	✔	✘	ExitToDebugger	
130		✔	FullGC	
131		✔	IncrementalGC	
136	✔	✘	SignalAtMilliseconds	
137	✔	✘	SecondsClock	
201–205	✔	✔	ClosureValue	*only for closures*
206	✔	✘	ClosureValueWithArgs	*only for closures*
210	✔	✘	At	*ContextPart, ≅ 60*
211	✔	✘	AtPut	*ContextPart, ≅ 61*
212	✔	✘	Size	*ContextPart, ≅ 63*
221/222	✔	✘	ClosureValueNoContextSwitch	*only for closures*
multiple	n/a	n/a	Fail	*deliberately failing*

Appendix E.

Key for Virtual Machine Structure Diagram


VM component,
Operating system
Data storage
Access
Access, not directly related to VM behavior
Data-loading access, no control behavior like GC.
Model boundary

Key for figure 4.1 and figure 4.2. Note that the former diagram uses a non-formal notation in both the lecture material [16] and the book it is based on [44]. Hence, to point out the abstracted parts, we chose to not use a formal notation either but rather use the same non-formal for the latter diagram.

Appendix F.

Code Examples for S1X

```
Behavior>>basicNew
  ”Primitive. Answer an instance of the receiver (which is a class)
  with no indexable variables.
  Fail if the class is indexable. Essential.
  See Object documentation whatIsAPrimitive.”

  <primitive: 70>
  self isVariable ifTrue: [↑ self basicNew: 0].
  ”space must be low”
  OutOfMemory signal.
  ↑ self basicNew ”retry if user proceeds”
```

Listing F.1: Squeak/Smalltalk source of `Behavior>>new` as of Squeak 4.3.

```
<primitive: 70>
33 <70> self
34 <D1> send: isVariable
35 <9B> jumpFalse: 40
36 <70> self
37 <75> pushConstant: 0
38 <E0> send: basicNew:
39 <7C> returnTop
40 <43> pushLit: OutOfMemory
41 <D2> send: signal
42 <87> pop
43 <70> self
44 <D4> send: basicNew
45 <7C> returnTop
```

Listing F.2: Squeak bytecode of `Behavior>>new` as of Squeak 4.3.

```
;;
;; sr = S1XRuntime
;; mr = MaxSqueakRuntime
;;
Prologue:
     jmp       L1
     nop
     nop
     nop
     call      +133940307       ; OPT2BASELINE-Adapter(R)
L1:
     enter     0x50, 0x0        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rbp, 0x50
     mov       [rsp - 8104], rax
     xor       r11, r11
     mov       [-159], r11
     xor       r11, r11
     mov       [rbp + 72], r11
Primitive:                      ; primitive 70
     mov       r11, [rbp + 96]
     subq      rsp, 0x10
     mov       [rsp], r11
     mov       rdi, [rsp]
     mov       rdi, [rdi + 16]
     nop                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     nop
     nop
     call      -9006170         ; sr.createTupleOrHybrid()[O]
     nop                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov       [rsp], rax
     subq      rsp, 0x10
     cmpq      rax, 0x0
     jz        L2: +8
     addq      rbp, 0x50
     leave
     ret       0x10

bytecode_01:                    ; push receiver
L2:
     mov       r11, [rbp + 96]
     subq      rsp, 0x10
     mov       [rsp], r11

bytecode_02:                    ; send: #isVariable
     mov       rsi, [-232]      ; aSymbol(isVariable)
     mov       rdi, [rsp]
     call      -9007353          ; sr.resolveAndSelectSqueakMethod()[O]
     nop                         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov       rdi, [rsp]
     call      rax
     nop                         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rsp, 0x10
     mov       [rsp], rax

bytecode_03:                     ; jumpFalse: 40 (L5)
     mov       r11, [rsp]
     mov       rax, [-260]       ; aTrue(true)
     addq      rsp, 0x10
     cmp       r11, rax
     jz        L3: +86           ; bytecode_04
     subq      rsp, 0x10
     mov       r11, [rsp]
     mov       rax, [-280]       ; aFalse(false)
     addq      rsp, 0x10
     cmp       r11, rax
     jz        L5: +256          ; bytecode_08
     subq      rsp, 0x10
     mov       rdi, [-296]       ; SpecialObject.SelectorMustBeBoolean
     call      -9005741          ; sr.getSpecialObjectReference()[O]
     nop                         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rsp, 0x10
     mov       [rsp], rax
     mov       rsi, [rsp]
     addq      rsp, 0x10
     mov       rdi, [rsp]
     call      -9007463          ; sr.resolveAndSelectSqueakMethod()[O]
     nop                         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov       rdi, [rsp]
     call      rax
     nop                         ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rsp, 0x10
     mov       [rsp], rax

bytecode_04:                     ; push receiver
L3:
     mov       r11, [rbp + 96]
     subq      rsp, 0x10
     mov       [rsp], r11

bytecode_05:                     ; push constant 0
     mov       r11, [-354]       ; 0, tagged int
     subq      rsp, 0x10
     mov       [rsp], r11
```

```
bytecode_06:                      ; send: #basicNew:
     mov      rsi, [-361]         ; aSymbol(basicNew:)
     mov      rdi, [rsp + 16]
     call     -9007523            ; sr.resolveAndSelectSqueakMethod()[0]
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rdi, [rsp + 16]
     call     rax
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq     rsp, 0x10
     mov      [rsp], rax

bytecode_07:                      ; return top
     mov      rdi, [rsp]
     mov      rax, rdi
     addq     rsp, 0x10
     mov      r11, [rbp + 72]
     mov      rdi, r11
     mov      rsi, rax
     mov      rdx, rdi
     cmpq     rdx, 0x0
     jz       L4: +84
     mov      [rbp + 24], rsi
     mov      [rbp + 8], rdi
     mov      rax, [-429]         ; mr
     mov      rdx, [rax + 24]
     mov      rdi, rax
     mov      rsi, rdx
     mov      edx, 0x0
     mov      [rbp + 16], rax
     call     -9014213            ; mr.getElement()[0]
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rdi, [rbp + 8]
     mov      [rdi + 40], rax     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rax, [rbp + 16]
     mov      rsi, [rax + 24]
     mov      rdi, rax
     mov      edx, 0x0
     nop
     nop
     call     -9014245            ; mr.getElement()[0]
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rdi, [rbp + 8]
     mov      [rdi + 32], rax
     mov      rsi, [rbp + 24]
Epilogue_L4:
L4:
     mov       rax, rsi
     subq      rsp, 0x10
     mov       [rsp], rax
     addq      rbp, 0x50
     leave
     ret       0x10

bytecode_08:                      ; push literal: OutOfMemory
L5:
     mov       rdi, [-516]        ; anAssociation(#OutOfMemory->OutOfMemory)
     call      -9014049           ; sr.getAssociationValue()[O]
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rsp, 0x10
     mov       [rsp], rax

bytecode_09:                      ; send: #signal
     mov       rsi, [-529]        ; aSymbol(signal)
     mov       rdi, [rsp]
     call      -9007714           ; sr.resolveAndSelectSqueakMethod()[O]
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov       rdi, [rsp]
     call      rax
     nop                          ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     subq      rsp, 0x10
     mov       [rsp], rax


bytecode_10:                      ; pop
     addq      rsp, 0x10


bytecode_11:                      ; push receiver
     mov       r11, [rbp + 96]
     subq      rsp, 0x10
     mov       [rsp], r11

bytecode_12:                      ; send: #basicNew
     mov       rsi, [-569]        ; aSymbol(basicNew)
     mov       rdi, [rsp]
     call      -9007762           ; sr.resolveAndSelectSqueakMethod()[O]
     nop
     mov       rdi, [rsp]
     call      rax
     nop
     subq      rsp, 0x10
     mov       [rsp], rax
```

```
bytecode_13:                    ; return top
     mov      rdi, [rsp]
     mov      rax, rdi
     addq     rsp, 0x10
     mov      r11, [rbp + 72]
     mov      rdi, r11
     mov      rsi, rax
     mov      rdx, rdi
     cmpq     rdx, 0x0
     jz       L6: +84           ; Epilogue
     mov      [rbp + 24], rsi
     mov      [rbp + 8], rdi
     mov      rax, [-635]       ; mr
     mov      rdx, [rax + 24]
     mov      rdi, rax
     mov      rsi, rdx
     mov      edx, 0x0
     mov      [rbp + 16], rax
     call     -9014451          ; mr.getElement()[0]
     nop                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rdi, [rbp + 8]
     mov      [rdi + 40], rax   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rax, [rbp + 16]
     mov      rsi, [rax + 24]
     mov      rdi, rax
     mov      edx, 0x0
     nop
     nop
     call     -9014483          ; mr.getElement()[0]
     nop                        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
     mov      rdi, [rbp + 8]
     mov      [rdi + 32], rax
     mov      rsi, [rbp + 24]

Epilogue:
L6:
     mov      rax, rsi
     subq     rsp, 0x10
     mov      [rsp], rax
     addq     rbp, 0x50
     leave
     ret      0x10
```

Listing F.3: S1X generated native code for Behavior>>new.

Colophon

This report was typeset by $\LaTeX\,2_\varepsilon$ with XƎTEX using KOMA-Script. The body text is set 11/14¼ pt on a 30¼ pc measure. The body type face is Hermann Zapf's *Palatino Linotype*. The listing type face is *DejaVu Sans Mono*, based on the *Vera* family by Bitstream, Inc. *—Tobias Pape*

ISBN 978-3-86956-266-7
ISSN 1613-5652