Supporting Multiple Languages in Virtual Machines

by

Jan Vraný

A thesis submitted to
the Faculty of Information Technologies, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Informatics

September 2010
Thesis Supervisor:

Doc. Ing. Vojtěch Merunka, PhD.
Department of Software Engineering
Faculty of Information Technologies
Czech Technical University in Prague
Kolejn 550/2
160 00 Praha 6
Czech Republic

Copyright © 2010 by Jan Vraný
Abstract and contributions

Programming language is one of the key tools used to develop software systems. Programming languages, even called “general purpose” language, differ in many aspects. One language might be more suitable for solving one sort of problems, whereas other might be better for solving another type of problems. Programming environment that enables programmers to use several programming language within one program, therefore, would ease and speed up a construction of such a program. Despite all the effort spent in past few years, the support for hosting multiple languages in modern programming environments is still limited.

In particular, we will discuss three problems that arising during the development and usage of the applications written in several programming languages:

- lack of debugging facilities for language interpreters based on an interpreter design pattern
- inability of debuggers to cope with programs written in several languages using several different language runtimes
- inflexibility of modern virtual machines to handle different language semantics in a efficient way

In this thesis we propose solutions to problems mentioned above. We describe an Smalltalk implementation of these solutions. In addition, we provide case studies and benchmarks that validates proposed approaches.

Keywords:
programming language, interpreter, virtual machine, debugger, metaobject protocol, Smalltalk, Ruby
As a collaborator of Jan Vraný and a co-author of his papers, I agree with Jan Vraný’s authorship of the research results as stated in this dissertation thesis.

..............................

Alexandre Bergel
Acknowledgements

This work could not have been completed without the support of a number of helpful people. First of all, I would like to thank to Mr. Vojtěch Merunka, whose lectures and enthusiasm has completely changed my view on programming. Without him, I would never discovered a wonderful world of Smalltalk. At the very beginning, his ideas were a great source of inspiration for me. He incited my interest in programming languages.

During my work, it was a great pleasure work with Alexandre Bergel. I have learned a great deal from him. I appreciate our long and valuable discussions and advices Mr. Berger gave me.

My special thanks goes to Mr. Claus Gittinger as well. His ideas were another source of deep inspiration. He has given me an opportunity to work on Smalltalk/X. Without his continuous support, I would have never finished this work. He was always ready to explain me all the complex low-level issues and gave me valuable hints, if not the solution, whenever I felt I got got lost. Great attitude for all the inspiring discussions and support.

In would like to thank to Mr. Michal Valenta. He has always done his best to provide me a pleasant environment for my research. I really appreciate it. I hope I will be able to repay.

Special thanks to my students Jan Karpíšek and Petr Špaček for their marvelous work on SMALLRUBY.

Finally, my greatest thanks to my family and friends whose support was of a great importance during all my research.
Contents

1 Introduction 1
  1.1 Understanding the Problem ................................. 2
    1.1.1 Multiple-language Applications ......................... 2
    1.1.2 Custom Language Interpreters .......................... 3
    1.1.3 Virtual Machines .................................. 3
  1.2 Hypothesis .............................................. 3
  1.3 Contribution ............................................ 4
  1.4 Thesis Outline ........................................... 5

2 Motivation 7
  2.1 Language Extensions ................................…… 7
    2.1.1 Traits ............................................... 8
      2.1.1.1 Implementation Issues .............................. 8
    2.1.2 Classboxes .......................................... 9
      2.1.2.1 Implementation Issues ............................. 10
    2.1.3 Selector Namespaces .................................. 10
      2.1.3.1 Implementation Issues ............................. 11
  2.2 Running Multiple Languages within a Single Virtual Machine 11
    2.2.1 #Smalltalk .......................................... 13
      2.2.1.1 Classes ....................................... 13
      2.2.1.2 Methods ...................................... 14
      2.2.1.3 Blocks ...................................... 14
    2.2.2 JRuby ............................................... 17
  2.3 Summary ................................................. 18

I Concepts 21

3 Debuggable Interpreter Design Pattern 22
  3.1 Introduction ............................................. 22
  3.2 Interpreting and Debugging Languages ...................... 23
5.3.3 Custom Lookup Routine .............................................. 50
5.4 Validation .............................................................. 51
  5.4.1 Method Privacy .................................................... 51
  5.4.2 Selector Namespaces .............................................. 51
  5.4.3 SMALLRUBY ....................................................... 52
5.5 Discussion ............................................................. 52
  5.5.1 Cache Effects ..................................................... 52
  5.5.2 Method Inlining ................................................... 53
  5.5.3 Multiple/Predicate Dispatch ................................. 53
  5.5.4 Performance and Memory Overhead ......................... 53
5.6 Related Work ........................................................ 53
  5.6.1 Smalltalk #doesNotUnderstand: ........................... 53
  5.6.2 JSR 292: Invoke Dynamic .................................... 54
  5.6.3 .NET DLR ......................................................... 54
  5.6.4 COLA ............................................................. 55
  5.6.5 JikesRVM ........................................................ 55
5.7 Summary ............................................................... 55

II Validation

6 The PERSEUS Framework .............................................. 57
  6.1 Using PERSEUS for Language Construction .................. 57
    6.1.1 Abstract Syntax Tree Construction ....................... 57
    6.1.2 Writing an interpreter ..................................... 58
  6.2 Graphical Tools Provided by PERSEUS ......................... 59
  6.3 Writing Tools on the Top of the PERSEUS Framework ...... 59
    6.3.1 Conditional Breakpoints .................................. 60
    6.3.2 Dynamic Analysis .......................................... 61
  6.4 Practical Applications of PERSEUS ............................ 64
    6.4.1 CELLSTORE/XQuery ....................................... 64
    6.4.2 IZAR .......................................................... 64
  6.5 Summary .......................................................... 64

viii
List of Figures

2.1 Examples of imports and refinements ............................................ 10
2.2 Example of two class extensions on a class String taken from [10] .......... 11
2.3 # Smalltalk classes – solid lines are inheritance relationships and dashed lines are #class pointers. ......................................................... 13
2.4 Example of the method with a block .................................................. 14
2.5 Pseudobytcde for the method presented in Figure 2.4 .......................... 14
2.6 Example of method with a block with non-local return ......................... 15
2.7 Pseudocode for the nested block of the method in Figure 2.6 .............. 15
2.8 Preudocode for the method in Figure 2.6 ......................................... 16
2.9 Evaluation of the method evaluateNonLocalReturn will result in a non-local return to already unwound block ....................................... 16
2.10 Non-local variable mutating methods .............................................. 17
2.11 Example of the Ruby code and its equivalent code in Java ................ 19
2.12 An exerpt of a bytecode generated by JRuby for the method foo from Figure 2.11 ................................................................. 20

3.1 The SmallScript interpreter. ......................................................... 23
3.2 A graphical user interface of a debugger. ......................................... 24
3.3 A step-over operation does not go into the recursion. ......................... 26
3.4 A step-into operation goes into the recursion. .................................. 26
3.5 Announcement class hierarchy. ..................................................... 27
3.6 Definitions of the debugger service. ............................................. 30
3.7 Ratio between a non-debuggable and a debuggable interpreters. The use of a just-in-time compiler (JIT) is denoted in black. ...................... 34

4.1 Overall architecture of unified debugger ........................................... 39
4.2 Debugger adapter stack ............................................................ 43
4.3 Unified debugger user interface showing all languages ..................... 44

5.1 Lookup object lookup algorithm ................................................... 49
5.2 Method lookup routine hard-coded to the virtual machine ............... 50

6.1 The SmallScript Workspace ..................................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Performance benchmarks of an MOP-enabled VM</td>
<td>69</td>
</tr>
<tr>
<td>7.2</td>
<td>Smalltalk/X System Browser showing the method in a namespace</td>
<td>75</td>
</tr>
<tr>
<td>7.3</td>
<td>Selector namespace aware lookup method</td>
<td>76</td>
</tr>
<tr>
<td>8.1</td>
<td>Dynamically computed superclass in Ruby</td>
<td>79</td>
</tr>
<tr>
<td>8.2</td>
<td>Compilation of a conditional expression</td>
<td>82</td>
</tr>
<tr>
<td>8.3</td>
<td>SMALLRUBY’s method lookup implementation</td>
<td>83</td>
</tr>
<tr>
<td>8.4</td>
<td>Selected Ruby Performance Benchmarks</td>
<td>87</td>
</tr>
</tbody>
</table>
1 Introduction

During the last decades, many programming languages have been designed, more or less successful. The increasing complexity of software still poses new challenges. Software engineers all over the world still need to cope with software composition and fast evolution. A programming language is a primary tool of a software developer. It is a tool used to express programmer’s ideas to a computer. Hence, the expressiveness and usability of the programming language is crucial in the software development process. After decades, new programming languages or various extensions of existing ones have still being developed. Each of them tries to point out drawbacks of the existing languages and provide new solutions that shall result in faster development of more reliable software.

At the same time, the applications written in several programming languages are not uncommon. The reasons for utilizing multiple languages within a single application vary: first, some parts of the application’s logic are sometimes much easily and accessibly expressed in a different language than the rest of it. Such a situation leads either to the creation of a relatively small scripted language that belongs to the category of the domain-specific languages [51] or to the usage of a general purpose scripting language such as Python or JavaScript. Second, it might be efficient in terms of time and programmers’ effort to develop parts of an application in different languages because of already existing reusable codebase, libraries or suitable frameworks. Third, a low-level language (usually C) routines are often embedded into other languages for performance reasons and finally, the utilization of several languages in one application may simply be the only technologically available option (e.g., when implementing triggers in Oracle database, Java or PL/SQL must be used even though the rest of the application is written in PHP). In other words, taking the best from each language helps application developers to fulfill user’s requirements better and faster.

An implementation of a programming language usually consists of two parts: a compiler and an execution runtime. The compiler’s role is to transform the source code of an application into a representation which is understood by its respective execution runtime: machine code, byte code or an abstract syntax tree. In some cases, the runtime executes the source code directly, which eliminates the need for the compiler (e.g., some UNIX shell interpreters for example).

The role of the runtime is then to perform the computation by interpreting the intermediate program representation created by the compiler. Most modern programming languages require non-trivial execution runtime: the virtual machine (VM).

In certain cases, there is no need for an execution runtime since the compiler generates machine code that runs on bare hardware. However, with the progressi of virtual machines technology and, at the same time, increasing complexity of standard libraries used in languages compiled into native code, the distinction between pure interpretation and a direct machine code execution is getting more and more blurred.
1.1 Understanding the Problem

1.1.1 Multiple-language Applications

Basically, there are three ways how to mix two or more languages in a single application. The first one uses an interpreter written in one of the languages to interpret the other languages. In such case, the invocation of a method written in the second language from a code written in the first language may be achieved by calling the interpreter with the function identification passed as an argument. In other words, the developer invokes the interpreter which, in turn, invokes the function written in another language.

This approach is usually used for the execution of small pieces of a code written in a specialized domain-specific language (DSL) and it is often based on the interpreter design pattern [29]. Its advantage is that it is easy to implement, modify and extend. However, its performance tends to be poor in comparison with other approaches.

The second way is to pass the execution to a native code. This approach is both efficient and flexible, however, in this case, the calling language requires a special kind of support for invoking of the methods written in other languages. In other words, the language must provide some mechanism of stepping out of its runtime, converting all necessary data (parameters and return value) and invoking the runtime of the other language. The *JavaConnect* [39] is an example of such an approach. Obviously, such an option is not always viable.

The third way is to reuse the existing runtime and let the compiler produce an intermediate code representation (usually bytecode) understood by that runtime. Semantics that is not available on the target platform is emulated.

Such approach is very effective in terms of language implementor’s effort. It usually provides fair performance and the integration of the two languages is almost seamless in both ways. However, it is suitable only for the languages with fairly similar basic notions, *e.g.*, this approach would probably be very difficult to use to integrate a prototype-based language with continuations and without any notion of a call stack with the runtime designed for the language based on a logic programming paradigm. Despite the difficulties caused by possible semantic mismatch between the VM and a given language, such an approach is quite popular. Over 300 languages have been implemented on the top of JVM [65], the Newspeak programming language reuses Squeak VM [30]. Microsoft uses his Common Language Runtime (CLR) to run C#, Visual Basic or F#. There are many reasons for choosing an existing one: a high reuse of what others already did, no need for tackling all the low-level platform specific issues and a possibility to use all the code that is already running on that VM.
1.1.2 Custom Language Interpreters

An interpreter design pattern is a popular way of implementing a small, non performance-critical language (usually a sort of DSL). Such an interpreter is easy to implement and maintain, even though the interpreter design pattern provides no support for debugging and tracing interpreted program. As the complexity of programs written in such an interpreted language grows, the necessity of the source-level debugger for the language become evident. Moreover, if the execution of the interpreted language interleaves with an execution of the interpreting language, e.g., if an application invokes function written in DSL which in turn calls back the application’s API routine, the need for a single debugger for both languages arises.

1.1.3 Virtual Machines

Virtual machines provide a useful abstraction both of the underlying hardware and operating system. On the other hand, they are usually designed for a specific language (or sort of languages). Parts of the language semantics is usually hard-coded in the VM, especially in the mature ones, for various reasons: it makes high-level layers simpler because of improved performance or security. For example, the way how the VM finds a proper method for a given method call – a method lookup algorithm – is usually implemented by means of a “send” or an “invoke” instruction. If the semantics built in the VM differs from the semantics of the language being run on a particular VM, it must be emulated at an object level. The reason is that current VMs provide no or little support for altering its built-in behavior. Moreover, modern VMs contain highly sophisticated optimizations to speed up a program execution. By emulating features at the object level, these optimizations could be hardly used.

1.2 Hypothesis

We need more open, language independent virtual machines.

Although all general purpose programming languages are theoretically equivalent i.e., they can express any program, in practice, they differ. Using the programming language is a way how human beings, programmers, communicate with computers. Some tasks are easier to solve in one language than in another, which always depends on a particular task and language. The applications are becoming more and more complex and the need for multi-language programming environment is evident. Since most modern programming languages use virtual machines, better support for multiple languages in virtual machines is highly desirable.

In past few years, a significant effort has been spent on running multiple languages on the top of mature virtual machines such as JVM or CLR. Despite all the effort, current support
for running multiple languages is still quite limited. We believe that the main problem is that the current virtual machines were designed specifically for one programming language.

**Thesis statement.** To build a truly multi-language programming environment, we need more open, language-independent virtual machine.

In order to achieve required flexibility, the virtual machine should either make no assumptions on language semantics it run on or, to provide mechanisms to override built-in semantics.

This is a difficult problem to solve. We need a virtual machine that can run multiple languages (possibly with completely different semantics) side by side. On the other hand, we want a virtual machine to provide suitable high-level abstractions that eases language implementation and, at the same time, we wish programs to run efficiently compared to an equivalent optimized C code. These two requirements are in a clash: how can the virtual machine provide suitable abstractions and optimize the code when it does not know anything about the code it runs?

In this thesis we focus on two aspects:

- **Construction of a debugger for the language interpreter based on an AST interpretation.** A direct interpretation of a program’s abstract syntax tree (AST) is a widely used approach to build a language interpreter. With an increasing number of codes executed by such an interpreter, however, the lack of advanced debugging tools makes further development and maintenance more and more time consuming and error prone. Therefore we focus on a symbolic debugger for the AST interpreter.

- **Definition of a metaobject protocol allowing the customization of a method-lookup algorithm.** Calling the method is the most common operation in a modern object-oriented code and thus, a method lookup is extensively optimized by the VM. The method lookup usually differs among programming languages. In order improve multi-language support in VMs, we focus on a metaobject protocol allowing the programmers to define their own method lookup algorithms while preserving the performance achieved by the optimizations at the VM level.

### 1.3 Contribution

The main contribution of the thesis is the following:

- **Debuggable Interpreter Design Pattern**
  The Debuggable Interpreter Design Pattern describes a programming language interpreter that offers debugging facilities. It augments the Interpreter pattern with few hooks in the visiting methods and employs a debugging service to model debugging operations (i.e., step-in, step-over or continue).
• **Perseus Framework**
  Perseus is a ready-to-use framework based on a debuggable interpreter design pattern that eases the construction of the language interpreters. First, it provides both a set of reusable classes to build an interpreter and an extensive set of tools for evaluating, debugging and profiling code written in a Perseus-based language.

• **Unified Debugger Architecture**
  A proposal for debugger architecture that is capable of debugging the programs written in multiple languages using several different language runtimes such as a VM or an AST interpreter. A proof-of-concept implementation of such a debugger validates the proposal.

• **Virtual Machine Level Metaobject Protocol for Customizing Method Lookup**
  It describes a simple metaobject protocol (MOP) [42] for controlling method lookup. It allows programmers to alter the method lookup semantics from outside the virtual machine. The method lookup MOP has been designed with respect to VM-level techniques used to optimize the method dispatch. Therefore, its impact on overall performance is in average unnoticeable.

• **Evaluation**
  Proposed approaches have been evaluated in a real-world programming environment – a Smalltalk dialect called Smalltalk/X and SMALLRUBY, an experimental Ruby implementation for Smalltalk/X VM. This thesis contains a brief description of changes we made to Smalltalk/X and an outline of an implementation of the SMALLRUBY. Performance benchmarks are also presented.

### 1.4 Thesis Outline

This thesis is structured as follows:

• Chapter 2 provides more detailed analysis of existing virtual machines and programming environments and presents examples of their limitations.

• Chapter 3 presents the debuggable interpreter design pattern.

• Chapter 4 describes an architecture of a unified debugger, a symbolic debugger that can debug homogeneous applications written in multiple languages running under different runtimes.

• Chapter 5 is focused on a VM-level metaobject protocol for the method lookup.

• Chapter 6 provides a brief overview of Perseus – a framework for language construction based on a debuggable interpreter design pattern described in chapter 3.
• Chapter 7 summarizes and discusses the changes made to the Smalltalk/X in order to evaluate ideas presented in previous chapters. It also provides benchmarks that support our claims about the performance.

• Chapter 8 briefly describes our experience with implementing Ruby on the top of Smalltalk environment and compares the performance of our Ruby implementation with other implementations.

• Finally, Chapter 9 concludes and outlines the ways of future research.
2 Motivation

Virtual machines for high-level object languages are usually complex pieces of the code. Their individual parts are tightly interlinked to each other, so any change in one part results in changes in one or more other parts. On the other hand, object-oriented languages have been still evolving. Software engineers all over the world are developing new programming languages, extending the current ones or trying to modify their semantics in order to solve problems occurring in practice. The evaluation of new ideas and approaches, however, can be very difficult – and often is. Proposed enhancements can require non-trivial changes to an existing virtual machine. As we mentioned, such changes are difficult to make, if even possible. A more open and extensible virtual machine would help both software engineers by providing a test bed for their ideas, and application developers by providing a flexible environment that could be easily adapted to meet their specific needs.

In this section, we will present five examples as an evidence that current virtual machine implementations are not flexible enough to handle useful language extensions or a completely new language. The first example (Section 2.1.1) shows traits, a new mechanism for code reuse. Section 2.1.2 presents classboxes – a module system for object languages supporting local refinements. The third example (Section 2.1.3) describes selector namespaces, an advanced module system for class extensions. Last two examples (Sections 2.2.1 and 2.2.2, respectively) deal with the implementation of new languages on the top of existing VMs: Smalltalk for Microsoft’s Common Language Runtime (CLR) and Ruby for the Java Virtual Machine (JVM).

2.1 Language Extensions

In object languages, a method lookup is a process that determines which method has to be executed in response to a message send. The method lookup algorithm usually takes a receiver (the object the message was sent to) and a method selector (an identification of the method, usually its name) and returns the method that is to be executed. If no matching method is found the error is triggered. For given language, the lookup algorithm is a part of its formal specification.

In dynamically typed languages such as Smalltalk the method lookup is performed by the virtual machine in run-time virtually whenever a message is being sent. In statically typed languages such as C++, the method lookup is performed in a compile-time by a compiler.

The method lookup algorithm is traditionally hard-coded in the virtual machine (or in the compiler), and thus it cannot be changed. In next two sections, we present two language extensions whose natural implementation implies the changes to the method lookup algorithm.
2.1.1 Traits

Traits [59, 57, 55] are a novel approach to deal with code composition and reuse. Many object-oriented languages provide single inheritance only i.e., a class may inherit its behavior from at most one class. Although the single inheritance is sufficient in most cases, in certain situations it is not expressive enough to factor out all common behavior shared by classes. Such situation leads usually in unwanted code duplication. Other languages such as C++ or CLOS provide multiple inheritance. Multiple inheritance, as implemented in C++, helps to avoid code duplication in certain cases but also brings about a wide range of problems, e.g., a problem of conflicting features (a “diamond problem” in particular [61, 16]), accessing overridden features, i.e., since features may be inherited from multiple superclasses, an thus to be able to use overridden feature a superclass must be explicitly named, which tangles the code with class hierarchy).

Traits attack all problems connected with classical single and multiple inheritance. They provide a mechanism to share the code among multiple class hierarchies and, more importantly, they provide tools to resolve conflicts. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. The classes are composed from a set of traits by specifying a glue code that connects the traits together and accesses the necessary state. One trait can be composed from other traits. In certain cases, traits effectively reduce the size of the code and therefore increase software maintainability [20].

Informally, when a message has been sent to an object with traits, the method is looked up as follows:

1. an object is asked for its class, being a current search class,
2. the search class is searched for a particular method,
3. if no method is found, search class’ traits are being searched,
4. if no method is found, the search class is asked for its superclass,
5. if there is no superclass, e.g., a root of class hierarchy is reached, the method is definitely not found and an error is triggered,
6. if there is a superclass, it becomes a current search class and lookup is restarted from Step 2.

2.1.1.1 Implementation Issues

Traits are included in recent versions of Squeak Smalltalk [62]. In Squeak, traits are used to compose essential classes such as Behavior, ClassDescription, Class and Metaclass.
A natural implementation of trait mechanism is relatively straightforward: first, it extends a class to maintain also a reference to the set of traits it uses and second, changes message a lookup algorithm to search also the traits [58]. Unfortunately, such changes to the virtual machine are, as we said before, very complex.

Squeak implementation of the traits is based on sharing instances of a CompiledMethod among the classes. When a trait is added to the class all its methods are inserted to its method dictionary. When a trait is removed from the class all its methods must be removed from method dictionary of the class.

A method-sharing approach has one major drawback: it must deal with recursive trait compositions, method renaming and exclusions. A method sharing is very complex and error-prone.

2.1.2 Classboxes

Whereas the traits target one software engineering task, a code composition and reuse, Classboxes [14, 12, 13, 9] are focused on another important software engineering problem – software evolution.

A classbox is a module system supporting local class refinements. Within a classbox, the classes, methods and variables can be defined. Each class, method and variable belongs precisely to one classbox, namely the one in which it is originally defined. A classbox may import classes from other classboxes making them visible in an importing classbox. Any class visible within a classbox may be imported by another classbox i.e., imports are transitive.

Imported classes can be locally refined – new methods can be added and imported methods can be redefined. Such refinements are visible only within such a classbox defining refinements and within classbox importing a refining classbox. In other words, if one classbox imports classes from another, it will see versions of classes possibly refined. In run-time, a method lookup starts in the classbox the very first method belongs to. For more details about the method lookup used in classbox-enabled environments see [7].

An example of imports and method refinements is shown in Figure 2.1. We define a class hello (within a classbox greetings-cb) having two methods: say-hello and say-good-bye. We also define a class app within a classbox application-cb whose method main simply uses an imported class hello to print greetings. When invoked from the classbox application-cb, a string “Hello! Good bye!” is printed. The classbox spanish-greetings-cb imports greetings-cb and refines the method say-hello of an imported class hello with a new implementation. Finally, we define a classbox spanish-application-cb. The invoking method main of the class app within that classbox will print “Hola! Good bye”.

SECTION 2. MOTIVATION

9
Figure 2.1: Examples of imports and refinements

### 2.1.2.1 Implementation Issues

Since classboxes define a new method lookup, an obvious implementation of classboxes imply a change in the method lookup algorithm in the virtual machine. Since such a change is very complex, both Squeak [8] and Java [11] classbox implementations are based on the reflection and bytecode manipulation.

### 2.1.3 Selector Namespaces

Selector namespaces [69, 60] is another type of a feature which helps to cope with software composition and evolution. Many object-oriented languages including Smalltalk, Ruby or C# provide class extensions - a mechanism to add or override a method for an already existing class which is defined in another module than in the extending one. As any module may extend a class in another module, naming conflicts are likely to occur. Selector namespaces provide a mechanism to solve such name clashes. A selector namespace defines a namespace for the methods: several methods with the same name but in
a different namespace can coexist in the same class. A selector namespace can import other namespaces.

During a method lookup routine, a class is first looked up for a method in same namespace as the sending method’s namespace. If no method is found, the search continues in an imported namespace, if any.

Figure 2.2 shows an example of two class extensions on class `String`. Both methods `asURL` can coexist in a class `String` since they are defined in different selector namespaces.

### 2.1.3.1 Implementation Issues

As in previous cases, an obvious implementation of the selector namespaces lies in the extension of the method lookup algorithm to consider selector namespaces.

Alternative implementation of the selector namespaces would use bytecode instrumentation and stack reification to look at the namespace of the sending method, and execute a proper code. Experiments showed that such implementation results in an approximately thousand times slower method dispatch compared to a normal dispatch as performed by the VM. Moreover, most of the VMs does not allow programmers to reify the stack.

### 2.2 Running Multiple Languages within a Single Virtual Machine

Virtual machines were traditionally designed with respect to the language they were designed for.
Nowadays, running other languages, usually those dynamically typed, than the one for which the virtual machine was designed for become more and more popular. The ability to run multiple languages within a single virtual machine has several advantages:

- Other language can be used for application scripting. Since both the application and the script use the same execution runtime the script is executed with (theoretically) the same efficiency as the main application.

- Application may use software components written in other languages – one can reuse what others have already done. Component reuse significantly reduces development costs.

- Newly designed languages can be implemented and evaluated much faster – there is no need to implement the whole virtual machine from the scratch.

Although modern industrial virtual machines have different design goals, they differ in details. Consider three virtual machines – Squeak virtual machine, designed for Squeak Smalltalk [62, 36], Java virtual machine [46], designed for Java language [32], and Microsoft .Net CLR [63], designed primarily for C# language [19]. They all have some common features including:

- automatic storage reclamation (garbage collection),

- stack-based,

- bytecode as the only code representation understood by the virtual machine,

- bytecode set containing the instructions for
  - loading/storing object references from/to an object,
  - pushing/popping object references onto/from a stack,
  - method invocations,
  - controlling execution flow i.e., jumping and branching.

These similarities and relative similarity between the languages enable compiling multiple languages into a target virtual machine’s bytecode, and thus they enable running multiple languages withing a single virtual machine. Moreover, recently designed virtual machines such as JVM, CLR or Smalltalk/X virtual machine [5], contain some support for hosting other languages [50] or the support is planned for the next release [37].

In the next section we will present examples showing that to have a common bytecode set is not enough to efficiently implement all features of the given (object) language.
2.2.1 #Smalltalk

#Smalltalk [18] is an implementation of Smalltalk compiler that generates CIL code [45] – an intermediate code understood by the Microsoft .NET CLR.

2.2.1.1 Classes

Both Smalltalk classes and metaclasses are represented by .NET classes. All Smalltalk classes are inherited from a Root class that is a subclass of the .NET System.Object class. Smalltalk instance variables correspond directly to the .NET instance variables (see Figure 2.3), and Smalltalk class instance variables are represented by .NET instance variables on the .NET class corresponding to the Smalltalk metaclass. Smalltalk class variables are represented by the .NET static variables and are in the .NET class corresponding to the class, not to the metaclass.

![Diagram showing Smalltalk and .NET class relationships](image)

Figure 2.3: # Smalltalk classes – solid lines are inheritance relationships and dashed lines are #class pointers.
2.2.1.2 Methods

The methods are compiled directly to the .NET CIL virtual functions. Most Smalltalk syntax is directly supported by CIL, but there are some features that are missing, e.g., literals and block closures. To support literals the Smalltalk pool object contains static variables that contain the arrays of all types of literals. These static variable arrays are initialized when the program starts.

2.2.1.3 Blocks

Block closures are more difficult to implement. All non-optimized blocks are compiled into specially created classes. Each class has instance variables only for those variables which the block references but does not define.

```
SomeClass>> find1: anObject
↑ someCollection detect: [:each | each = anObject] ifNone: [1]
```

Figure 2.4: Example of the method with a block

For example, the compilation of the method illustrated in Figure 2.4 will create two classes for two blocks. The first block references but does not define `anObject`, therefore `anObject` will be an instance variable of the new block class. The second block has no external references so its .NET class will not have any instance variables. Code generated for that method is outlined in Figure 2.5.

```
SomeClass.find1(anObject) {
  push instance variable someCollection
  push argument anObject
  create & push instance of SomeClass.Find1Block1
    with anObject as the parameter for the constructor
  create & push instance of SomeClass.Find1Block2
    with no parameters
  call virtual function for #detect;ifNone:
  return
}
```

Figure 2.5: Pseudobytecode for the method presented in Figure 2.4

**Blocks with non-local returns.** The blocks that have non-local returns are even more difficult, since the .NET CIL has no support for the non-local returns. #Smalltalk uses
the exceptions to simulate non-local returns. When a method having a non-local return is entered, the method generates a number that is used by the exception that is thrown. When a return exception is thrown with such generated number, the method handles that exception and returns back its value. If the generated number does not match then the return exception is passed to the outer handlers. Since the exceptions are used to simulate non-local returns, non-local returns are much slower in the #Smalltalk than in other Smalltalks.

An example of method with non-local return is shown in Figure 2.6. The code generated for the inner block with a non-local return is depicted in Figure 2.7. Finally, the code generated for the method find2: is shown in Figure 2.8

```plaintext
SomeClass>> find2: anObject

someCollection do: [:each | each = anObject ifTrue: [↑ anObject]].
↑ nil
```

Figure 2.6: Example of method with a block with non-local return

```plaintext
SomeClass.Find2Block1.ValueBlock1.value() {
    create & push instance of NonLocalReturnException with
    anObject instance variable and
    tag instance variable
    as parameters for the constructor
    throw
}
```

Figure 2.7: Pseudocode for the nested block of the method in Figure 2.6

**Non-local return to unwound context.** Consider the code shown in Figure 2.9. The first method (SomeClass >> nonLocalReturn) simply returns the block whose evaluation will result in a non-local return to the method. The second method (SomeClass >> evaluateNonLocalReturn) simply calls the first method to obtain the block and then the block is evaluated. As we said before, evaluation of the block will result in the non-local return to the context which is already unwound (method SomeClass >> nonLocalReturn created the block and terminated returning the block in the form of return value).

In original the Smalltalk 80 implementation, this was not an error; instead, a normal block-return was performed to the value-sender\(^\text{1}\).

A #Smalltalk compiler cannot easily handle non-local returns to already unwound context in a Smalltalk-80 way. In the #Smalltalk, such returns will result in an unhandled exception.

\(^{1}\)current Smalltalk implementation treats it as an error and the virtual machine raises an exception.
SomeClass.find2(anObject) {
  try {
    push instance variable someCollection
    create & push instance of SomeClass.Find2Block1 with
      anObject as the parameter for the constructor
    create & push instance of SomeClass.Find2Block1.ValueBlock1 with
      anObject and
      generated tag
      as parameters for the constructor
    call virtual function for #detect:ifNone:
    return
  } catch (NonLocalReturnException ex) {
    if tag of caught exception is equal to tag generated by this method
      then
        push returnValue stored in exception ex
        return
      else
        push exception ex
        rethrow
  }
}

Figure 2.8: Preudocode for the method in Figure 2.6

of the type NonLocalReturnException, because once the exception has been being raised, there is not any exception handler for the given tag on the stack.

SomeClass >> nonLocalReturn
↑ [ ↑ 1 ]
SomeClass >> evaluateNonLocalReturn
↑ self nonLocalReturn value

Figure 2.9: Evaluation of the method evaluateNonLocalReturn will result in a non-local return to already unwound block

**Mutating non-locally accessed variables.** Since Smalltalk dissociates a block-creation and block-evaluation, non-local variables accessed by the block can change in the meantime. Consider the methods presented in Figure 2.10. Both methods should return an integer of the value of 40.
The #Smalltalk compiler cannot handle mutating variables properly, because the values of the variable \( x \) within the method and within the block are stored on different physical locations. The value of the \( x \) variable accessed by the method is stored in its activation record, whereas the value of \( x \) variable accessed by the block code is stored as an instance variable in a corresponding block-object. Since the values of non-locally accessed variables are copied to the block-objects in the block-creation time, the values may differ in the evaluation-time.

```ruby
SomeClass >> mutation1
| x b |
x ← 10.
b ← [ x * 2 ].
x ← 20.
↑ b value.

SomeClass >> mutation2
| x b |
x ← 10.
b ← [ x := 40].
b value.
```

Figure 2.10: Non-local variable mutating methods

### 2.2.2 JRuby

JRuby [21] is an implementation of the Ruby [49] programming language for the Java Virtual Machine. As in version 1.5.1, it is Ruby 1.8 compliant. Although both languages – Java and Ruby – are called pure object oriented, there are fundamental differences:

- In Ruby everything is an object, \( i.e., \) there is no notion of the “primitive” data type like `int` as in Java.
- Ruby is dynamically typed, \( i.e., \) the type of an expression is generally not known at the compile time, unlike in Java.
- Ruby is designed to be agile and productive. It contains many useful shortcuts. For example: Ruby supports methods with variadic arguments which are not supported in Java.
- Ruby features mixins [16] to share a code among different class hierarchies whereas Java supports only single inheritance.
Since a JVM’s intermediate language is not flexible enough to cover all semantics of Ruby directly, missing features are emulated at the object level.

**Method calls.** To implement Ruby method call semantics properly, JRuby uses an indirect object-level simulator object. Consider the Ruby code shown in Figure 2.11. First method – the `foo` – simply calls the method `bar` passing its arguments. Instead of using JVM’s “invokevirtual”, JRuby compiles a call to a *call site simulator object* (Figure 2.12). To realize a Ruby method call in JRuby, the call-site simulator object must be loaded onto a stack (line 25-26 in Figure 2.12). The call-site object is maintained for each method call and persists through the multiple invocations. The call-site object contains static information about the method call such as the name of the method being called, the number of arguments, etc. Once the call-site object is loaded, a `call` method of the call-site object is invoked. The `call` method looks up the method in the given receiver and uses a Java reflection API to invoke the method.

A main drawback of this solution is its performance. The Ruby code from Figure 2.11 is 30-40 times slower than the corresponding java code running on the same virtual machine.

### 2.3 Summary

In this section, we presented five examples where the current virtual machines fail due to lack of extensibility. Although all presented problems can be, and they are, overcome using various tricks or simulator objects, more extensible virtual machine would make the implementation easier and more straightforward. Moreover, such workaround tends to be slow.
Ruby

```ruby
1 class RubySends
2 2 def foo(o1)
3     return bar(o1);
4 end
5 6 def bar(o1)
7     return baz(o1);
8 end
9 10 def baz(o1)
11     return o1;
12 end
13 14 def self.main()
15     o1 = Object.new()
16     s = RubySends.new()
17     i = 0;
18     while i < 1000000 do
19         s.foo(o1);
20         i = i + 1;
21     end
22 end
23 24 end
25 26 RubySends.main()
```

Java

```java
1 public class JavaSends {
2 2 public Object foo(Object o1) {
3     return this.bar(o1);
4 } 6 public Object bar(Object o1) {
7     return this.baz(o1);
8 } 10 public Object baz(Object o1) {
11     return o1;
12 } 14 public static void main(
15     String[] args) {
16     Object o1 = new Object();
17     JavaSends s = new JavaSends();
18     int i = 0;
19     while (i < 1000000) {
20         s.foo(o1);
21         i = i + 1;
22     }
23 } 25 } 26 }
27 28 RubySends.main()
```

Figure 2.11: Example of the Ruby code and its equivalent code in Java
Figure 2.12: An excerpt of a bytecode generated by JRuby for the method foo from Figure 2.11
Part I

Concepts
3 Debuggable Interpreter Design Pattern

3.1 Introduction

A *design pattern* is a general repeatable solution to a commonly occurring problem in a software design. It is a description (or a template) of the way how to solve a problem. It can be used in many different situations. Design patterns gained their popularity after Gamma, Helm, Johnson, and Vlissides has compiled and classified what had been recognized as common pattern [29, 28].

The interpreter and visitor design patterns are usually described in terms of interpreting grammars. For a given language, they define a representation for its grammar along with an interpreter sentences in the language [28]. Whereas the ability of the visitor and the interpreter patterns to define programming language interpreters is widely recognized [22, 6, 47], no approaches to facilitate the realization of a debugger is currently available, as far as we know.

The *Debuggable Interpreter Design Pattern* describes a programming language interpreter that offers debugging facilities. It augments the Interpreter pattern [28] with some hooks in the “visiting” methods and it employs a debugging service to model operations (i.e., step-in, step-over, ...).

The following chapter then discusses, first, the description of the debuggable interpreter pattern, and, second, the illustration example using SmallScript, a subset of JavaScript.

Section 3.2 discusses the challenges in implementing a debugger. Section 3.3 presents a debuggable interpreter pattern and its illustration using SmallScript, a minimal procedural language. Section 3.4 shows some properties of the pattern. Section 3.5 provides a brief overview of the related work. Section 3.6 concludes by summarizing the presented work.
3.2 Interpreting and Debugging Languages

3.2.1 The SmallScript Interpreter

SmallScript is a JavaScript-subset interpreter written in Smalltalk/X. SmallScript contains usual language constructions to define variables and functions. As an illustration, the following code describes the factorial function:

```javascript
function fact(i) {
    if ( i > 0 ) {
        return i * fact(i - 1);
    } else {
        return 1;
    }
}
var a;
a = 6;
fact(a);
```

Figure 3.1 provides an excerpt of a visitor-based interpreter for SmallScript and presents the body of the `visitAdditionNode` and `visitAssignmentNode` methods. An addition is realized by running the visitor on the left operand, then on the right operand, and finally returning the sum of these two values. An assignment is realized by running the visitor on the value of the assignment, and then storing the value in the context of the interpreter.

---

1In this chapter, SmallScript refers to a simple demonstration language and has nothing in common with SmallScript language developed by D. Simmons [60]

2We adopted the ST/X syntax in UML diagrams to present a homogeneous notation
The interpretation of the SmallScript programming language is realized through a direct application of the Interpreter and Visitor design pattern [28]. Figure 3.2 shows a debugging session involving the piece of code given above.

![Figure 3.2: A graphical user interface of a debugger.](image)

### 3.2.2 Realizing a Debugger

A debugger is a tool that is used for testing and debugging programs. Typically, the debuggers offer sophisticated functionalities such as running a program step by step, stopping (pausing the program to examine the current state) in some cases using a breakpoint, and tracking the values of some variables.

The interpreter maintains several registers such as the program counter (an instruction pointer) and the stack pointer. These registers represent the state of execution of the process defining the interpreter state.

Advanced debugging environments (e.g., Smalltalk/X, VisualWorks [67], Dolphin Smalltalk [26] enable several debuggers and interpreters for the same program code to coexist. Operations such as opening a new debugger from a debugger, debugging two different pieces of the same program, or debugging a multi-threaded program can be performed.

Whereas the state of an instruction-based interpreters is implied in a set of registers, recursive function invocations define the state of a visitor-based interpreter. The state of the interpreter is determined by the set of the function activation records, contained in the method calls stacks. A local context visualization is achieved by a sophisticated reflective
feature, such as a stack reification, which can result in a lack of performances or raise technical issues difficult to address.

A visitor-based interpreter allows the breakpoints to be set and it offers a mechanism to perform debugging operations such as a step-by-step instruction execution.

### 3.3 The Debuggable Interpreter Pattern

This section describes a general approach to realize and implement a debugger for a visitor-based interpreter. It augments the visitor interpreter with a set of hooks inserted in the `visit*` methods. As a part of the Interpreter design pattern, the dynamic information needed for a program interpretation is stored in a context. Debugging operations such as `step-into`, `step-by`, and `continue` are offered by a debugging service.

#### 3.3.1 Debugging Operations

Before describing the Debuggable Interpreter design pattern, it is important to explain the considered operations, since the debuggers come traditionally with their own set of definitions.

**Setting breakpoints.** A **breakpoint** is a signal that tells the debugger to temporarily suspend the execution of a program. A breakpoint is associated to a node in an abstract syntax tree. The same program can contain several breakpoints.

When the interpretation of an AST reaches a breakpoint, an interactive session begins, for which the operations described below can be invoked. The state of the debugger is modeled by both a context and by a reference to a particular node in the AST called the **current node**.

Breakpoints are set by the user through an interface. For example, right clicking on the interface presented in Figure 3.2 pops up a menu which offers a ‘set breakpoint’ entry.

**Step-over.** A **step-over** operation consists in moving to the following node in the AST after having interpreted the current node. The current node is then positioned on this new node.

Figure 3.3 illustrates a step-over operation. The current node is `fact(a)`. By performing a step-over operation, the current node is the `print(fact(a))` node.

**Step-into.** A **step-into** operation consists in moving to the next node in the AST according to the application control flow. This operation differs from the `step-over` by entering the recursion.
A step-into operation is illustrated in figure 3.4. In this situation, the interpreter halts at the first node in the recursion which is $i > 0$.

**Continue.** The execution of an application can be resumed by *continuing* it.

**Terminate.** The program execution can be prematurely ended with the *terminate* operation. As a consequence, no subsequent nodes are evaluated and allocated resources such as context stack and the interpreter are freed.

### 3.3.2 Events Emitted by the Interpreter

A number of event emissions can occur during a program interpretation. These events, called *announcements*, reflect particular situations such as reaching the of a breakpoint, accessing the context or entering or exiting the function. The latter ones are not necessary, but their emission eases the construction of debugging tools as it can be seen later in section 6.3. Figure 3.5 presents the announcement class hierarchy.
3.3.3 Hooks in the Visitor

The flow of a program interpretation stems from the invocation order of the visiting methods. Suspending and resuming the interpretation flow and capturing the recursion are the primitives for the debugging operations.

In the rest of this section, the interpreter class is called Interpreter. As described in Figure 3.1, it implements the visiting methods.

Hooks need to be inserted in the Interpreter class to enable step-over and step-in operations. These hooks inform a debugger service which part of the code is being executed by the announcements. As explained in Section 3.3.6, the service is a placeholder for debugging operations. The method Interpreter >> announceTracepointInterrupt enables the interpretation of an AST to be driven by a service:

1. `Interpreter >> announceTracepointInterrupt`
2. `announcer announce: (TracepointAnnouncement new)`

Those two methods have to be invoked when visiting the nodes. A visit: method maintains the current node reference in a context:

1. `Interpreter >> visit: aNode`
First, \texttt{visit: aNode} gets the reference of the previous node from the current activation context. This reference is used to set the current node back when \texttt{visit: aNode} has been completed. Then the interpreter notifies the new current node to the context. This new current node is the node being traversed. The interpreter runs over this node using a double dispatch.

The reference of the current node acts as an instruction execution pointer. It clearly identifies the current execution location.

Instead of directly performing a double dispatch, the \texttt{visit: aNode} must be used. For example, in the method \texttt{visitAdditionNode: aNode}, the recursion is obtained from invoking \texttt{visit: aNode}:

\begin{verbatim}
Interpreter >> visitAdditionNode: aNode
    left ← self visit: aNode left.
    right ← self visit: aNode right.
    self announceTracepointInterrupt.
↑ left + right
\end{verbatim}

Each \texttt{visit*} method must perform a call to \texttt{announceTracepointInterrupt} after traversing all branches and before synthesizing the result.

Compared with the code shown in Figure 3.1, this new version of \texttt{visitAdditionNode: aNode} make the interpreter be aware of breakpoints. When a breakpoint is reached, the execution of the interpreter is suspended. Subsequent subsections illustrate how the breakpoints and debugging modes are modeled.
3.3.4 Context Definition

Associations between the variables and values are stored within a context object [28]. The debuggable interpreter pattern augments this context with dynamic information related to the parent context and the current node under the execution. Each function invocation creates a new context.

The class `InterpreterContext` contains three variables: `sender` referring to the parent context, `currentNode` for the node in the abstract syntax tree, and `returnReached` indicating if the return node has been reached or not. The interpreter should not evaluate subsequent nodes once a return node has been reached. Typically, this occurs when a return statement is interpreted. The method `visitReturnNode: aNode` is therefore defined as follows:

```
Interpreter >> visitReturnNode: aNode
| value |
value ← self visit: expression.
self announceTracepointInterrupt
context returnReached: true.
↑ value
```

and the method `visit: aNode` must suppress the evaluation in case the return has been reached.

3.3.5 Separate Control Flow

An interpreter must be embedded in a thread. This is necessary for several reasons:

- Multiple execution of the same program enables a debugger to launch another debugger. Although it is not essential, this feature leads to a better comfort during the debugging.
- If the executed program fails, it should not impact the enclosing application environment. The interpreter cannot run in the same control flow as the programming environment. The program under the execution and the debugger cannot be executed in the same thread.

The `Interpreter` class defines a variable `process` and an `evaluate: aNode` method to trigger an execution:

```
Interpreter >> evaluate: aNode
| value semaphore |
semaphore ← Semaphore new: 0.
```
context ← InterpreterContext new.
process ← ([value ← self visit: aNode] newProcess)
    addExitAction:
        [semaphore signal.
        process ← nil].
        process resume.
    semaphore wait.
↑ value

The `evaluate: aNode` method creates a semaphore. This is necessary to block the execution of the interpreter. A new context is created. Then a new thread (called process in the Smalltalk/X terminology) is created, intended to execute the code `value ← self visit: aNode`. Independently, if an exception is raised or not, once completed, the exit action is triggered, which releases the semaphore.

### 3.3.6 Debugging Service

The actual debugging features are realized by a *debugger service*. This service implements the debugging operations such as step-into, step-over and continue. Figure 3.6 provides an overview.

![Debugging Service Diagram]

A control flow of an application is halted when it reaches a breakpoint, which identifies a location in the program. With the debuggable interpreter pattern, a breakpoint is identified as a node in an abstract syntax tree.
When a breakpoint is reached, the interpreter enters an interaction mode, which allows the user to perform further operations. The mode in which the service is set reflects the operation currently performed. Each mode represents a debugging operation. To keep this section concise, we consider only 3 modes: continue, step-into, and step-over.

The service maintains a list of breakpoints accessible by the modes. A breakpoint is added to the service by the user through the user interface.

The methods `continue`, `stepInto`, `stepOver` defined on `DebuggerService` are delegated to the current mode:

```
1 DebuggerService >> continue
  mode continue

2 DebuggerService >> stepInto
  mode stepInto

3 DebuggerService >> stepOver
  mode stepOver
```

The `tracepointReached` and `run` methods are used to steer the process associated to the interpreter. `tracepointReached` is invoked by the mode when a breakpoint is reached. This method simply suspends the process associated with the interpreter:

```
1 DebuggerService >> tracepointReached
  interpreter suspend

2 Interpreter >> suspend
```

**Setting breakpoints.** A debugger service maintains a list of nodes representing the breakpoints in the program.

```
1 DebuggerService >> addBreakpointOn: aNode
  self addBreakpoint: (Breakpoint onNode: aNode)

2 DebuggerService >> addBreakpoint: aBreakpoint
  breakpoints ifNil: [ breakpoints ← OrderedCollection new ].
  breakpoints add: aBreakpoint.

3 DebuggerService >> isBreakpoint: node
  breakpoints ifNil: [ ↑ false ].
  ↑ breakpoints anySatisfy:
```
The method `addBreakpointOn:` is invoked by a debugger user interface. The `aNode` parameter corresponds to the node in the abstract syntax tree that should halt the program interpretation.

**Continue mode.** The continue mode is the initial mode of the debugger service. When the debugger service is in the continue mode, the program is executed until a breakpoint is reached. In that case, the interpreter thread is suspended, and a debugger opened. The service can either switch to a step-into or step-over mode, or a continue mode. The two methods defining this mode are:

```smalltalk
ContinueMode >> tracepointInterrupt
(debuggerService isBreakpoint: interpreter context currentNode)
ifTrue: [ debuggerService tracepointReached ]

ContinueMode >> continue
debuggerService runInterpreter
```

**Step-into mode.** When the debugger service is in the step-into mode, the program interpretation is stopped (and the debugger is opened) once `tracepointInterrupt` is invoked.

```smalltalk
StepIntoMode >> tracepointInterrupt
debuggerService tracepointReachedFor

StepIntoMode >> stepInto
debuggerService runInterpreter
```

**Step-over mode.** When the debugger service is in the step-over mode, stepping does not follow recursions and method calls. The `StepOverMode` has a `context` variable. This variable captures the state of the current interpretation. It is initialized when the debugger service switches to this mode.

The two methods defining this mode are:

```smalltalk
StepOverMode >> tracepointInterrupt
```
If the current context and the current node match the ones referenced by the step-mode, then the debugger switches for a step-into mode, so that the execution will be halted on the node that follows the “step-overed” one.

3.4 Discussion

**Coexisting debuggers.** Since multiple interpreters can interpret the same program, several debuggers may be active at the same time. Although this feature is not considered as a priority for the Java debugger [4], it greatly enhances the debugging activity.

**Breakpoints.** The debuggable interpreter pattern emits a breakpoint signal when the control flow reaches a particular node in the abstract syntax tree. Note that this definition of a breakpoint might slightly diverge from a widely spread debugger such as GDB [2] where a breakpoint signal can be triggered when the control flow reaches a particular line of the source code. However, this behavior may be easily implemented, as long as the AST nodes keep track of their position in the source code.

**New operations.** New debugging operations can be easily added by subclassing the `DebuggerMode` and adding the corresponding methods in the `DebuggerService`. For example, the mode holding the condition to enable the interpretation can be implemented in the class `ConditionalInterpretationMode` in which the method `tracepointInterrupt` checks for the conditional expression.

**Speed and memory overhead.** We can be also interested in a speed and memory consumption overhead. The table below shows the time in millisecond to execute the factorial function with the debuggable interpreter design pattern (DIDP) and the classical visitor pattern (VDP). These figures are obtained while disabling the just-in-time compiler (JIT).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>DIDP (ms)</th>
<th>VDP (ms)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac(100)</td>
<td>5.0</td>
<td>2.0</td>
<td>2.50</td>
</tr>
<tr>
<td>fac(200)</td>
<td>8.0</td>
<td>5.0</td>
<td>1.60</td>
</tr>
<tr>
<td>fac(400)</td>
<td>17.0</td>
<td>12.0</td>
<td>1.41</td>
</tr>
<tr>
<td>fac(900)</td>
<td>38.0</td>
<td>20.0</td>
<td>1.90</td>
</tr>
<tr>
<td>fac(10000)</td>
<td>973.0</td>
<td>540.0</td>
<td>1.80</td>
</tr>
<tr>
<td>fac(70000)</td>
<td>22774.0</td>
<td>19722.0</td>
<td>1.15</td>
</tr>
</tbody>
</table>
Figure 3.7: Ratio between a non-debuggable and a debuggable interpreters. The use of a just-in-time compiler (JIT) is denoted in black.

Figure 3.7 shows the overhead of the DIDP with the VDP for each factorial expression. It also measures the benefit of having a JIT. The black bar indicates a measurement obtained with the JIT enabled; whereas the value denoted by a white bar is obtained with the JIT disabled.

As a result, we can see that the ratio is asymptotic to 1. This means that the cost of the debuggable interpreter design pattern compared the classical visitor one is negligible for deep recursion.

The table below compares the memory consumption of the DIDP with the VDP. The total number of created objects is shown for each factorial expression.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>DIDP</th>
<th>VDP</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac(100)</td>
<td>102</td>
<td>102</td>
<td>1</td>
</tr>
<tr>
<td>fac(200)</td>
<td>202</td>
<td>202</td>
<td>1</td>
</tr>
<tr>
<td>fac(400)</td>
<td>402</td>
<td>402</td>
<td>1</td>
</tr>
<tr>
<td>fac(900)</td>
<td>902</td>
<td>902</td>
<td>1</td>
</tr>
<tr>
<td>fac(10000)</td>
<td>10002</td>
<td>10002</td>
<td>1</td>
</tr>
<tr>
<td>fac(70000)</td>
<td>70002</td>
<td>70002</td>
<td>1</td>
</tr>
</tbody>
</table>

During the evaluation of the factorial, contexts are created for each recursion, plus the mode and a service. This table shows that the DIDP does not occurs any memory overhead.
The computer used for this experiment is an Intel Pentium M 1.6GHz, Linux (kernel 2.6.20.1), 512MB RAM, ST/X 5.2.8.

3.5 Related Work

Scripting debugging. Marceau et al. [48] designed a language for a scriptable debugger. The purpose of this debugger is to automatize sequences of debugging operations which can be laborious to repeat manually. A classical sequence can lie in setting a breakpoint, examining values of some variables and resuming the execution. Debugging an application generally may imply to repeat this sequence many times in order to find a single bug. Scripting a debugger helps to automate such a task.

Similarly, to the debuggable interpreter pattern, the scriptable debugger provides the primitives to capture the essential functionality of a debugger: observing a program’s state, monitoring its control path, and controlling its execution. This is achieved by explicit commands embedded in the program. In order to be debugged, a program must contain explicit trace points. This is a major difference with the debuggable interpreter pattern for which program does not need to be annotated.

Trace library. Hofer et al. [34] proposed a backward-in-time debugger. The Unstuck debugger allows one to navigating back to the history of the application. Their implementation uses of a trace library to collect the events and reconstruct the states. To generate events (a method invocation, variable access and a method return), the methods are instrumented using ByteSurgeon [24], a high-level library to manipulate the method bytecode.

Unstuck assumes that a program is interpreted by a virtual machine, whereas the debuggable interpreter design pattern relies on an interpretation driven by a visitor.

AST instrumentation. The Relational Meta-Language (RML) [53] is a language for writing executable Natural Semantics specifications. It is used to formally specify programming languages such as Java, Pascal, and MiniML. The RML debugger is based on an abstract syntax tree instrumentation to capture and record particular events. A post-mortem analysis tool is then provided to walk back and forth in time, display variable values, and execution points.

The AST is instrumented with debugging annotation related to the trace generation. From its design, the programming environment of RML is limited to one single debugger per session. Our approach allows several debugger to coexist.

Grammar weaving. Wu et al. [70] claims that the debugging is a concern that crosscuts a domain specific language specification. They propose to use AspectJ [1] to weave the
debugging semantics into the code created by a parser generator.

Their work is restricted to programming languages that are translated into a general purpose language. Our approach is different, since it assumes a program interpretation through a visitor and interpreter design pattern.

3.6 Summary

This chapter presents a general approach for implementing and realizing a debugger for a visitor-like interpreter. It extends a visitor with a set of hooks embedded in the visiting methods. The context primarily used to hold variables bindings has been extended with a reference to a parent context and keeps a reference to the node currently being interpreted. A debugger service models different operation available by means of a set of the mode.

Easy to implement, coexistence of multiple debuggers, and open to new debugging operations are the benefits of the Debuggable Interpreter Design Pattern.
4 Debugger Architecture for Multi-Language Environments

4.1 Introduction

The debugging code composed of chunks written in different languages brings about difficulties for developers, as debuggers usually support only one language. For instance, in case of a Java application that uses a native class (i.e., invokes embedded C functions). When debugging this application, developers have three basic options. First, they may use a Java debugger, which means they will be able to inspect input parameters and the return value of any embedded C function but they will not be able to debug the bodies of the C functions themselves. Second, they may use a C/C++ debugger – that way they will be able to debug the bodies of C functions as well but they will also have to step through the layers of a virtual machine implementation code when debugging Java code. Finally, they may decide to use the Java debugger to debug Java code and the C/C++ debugger to debug embedded C code but they will have to manually start/suspend each debugger when entering the code written in the respective language and they will have to merge information provided by the two user interfaces themselves which is hardly convenient or effective.

To sum it up, when debugging multiple-language applications, debuggers fail to provide programmers with the same quality of the user experience that is common for single-language applications debugging. Consequently, debugging of multiple-language applications is more resource-costly and error-prone\(^1\). Naturally, a debugger providing the same quality of user experience regardless the fact whether the application is single- or multiple-language is highly desirable.

The aim of this chapter is to propose architecture of such a debugger. Namely, its contributions are as follows: first, description of a flexible source-level debugger architecture facilitating integration of multiple single-language debuggers into a single unified debugging environment that allows debugging multi-language applications in their entirety and second, a proof-of-concept implementation that allows debugging of applications written in Smalltalk, XQuery and JavaScript.

The structure of this section is the following: Section 4.2 provides brief description of debugger implementations, Section 4.3 describes the proposed solution, Section 4.4 compares our solution with the related work and Section 4.5 concludes.

\(^1\)Which is especially funny since the debugger is supposed to ease and speed up removal of bugs – not to complicate it and slow it down.
4.2 Debuggers and Their Implementations

Debuggers are developers’ tools which ease the finding and removal of bugs in the code. Typically, debuggers allow the programmer to stop the debugged program’s execution when it reaches one of the previously designated statements in the code (breakpoints), execute the program’s expressions and statements step by step manner in their respective order (we call this a debugging operation) and inspect the program’s state, i.e. variables on the stack and on the heap (execution state).

There are two basic means of collecting data about which expression or statement is currently under the execution: code instrumentation and events emitted by an operating system, virtual machine or an interpreter [44].

When using the code instrumentation, the application’s code is augmented to contain the method calls that inform their recipient about the currently executed expression. The code can be instrumented statically—by the compiler or some kind of a code instrumentation tool—or dynamically—when it is being loaded into the memory or even before its first execution [33].

The second approach differs from the first one in a fact that the events are emitted by the interpreter, virtual machine or operating system [70]. In other words, debugging-related instructions are the part of the interpreter, virtual machine or operating system, not of the application itself.

To map program’s execution state to its source code, debuggers make use of debug symbols. Debug symbols are metadata enabling the debugger to gain additional information about the code—the names of the methods and variables, mapping from the instruction to the lines of code etc. They are generated by the compiler and can be distributed either with the compiled code or in a separate file.

4.3 Solution

This section describes the architecture of a unified debugger – a new debugger implementation capable of debugging a multi-language application.

4.3.1 Overview

The architecture of the unified debugger is a generalization of an infrastructure built for a debuggable interpreter design pattern shown in Chapter 3. Figure 4.1 shows core components of the debugger.

DebuggerAdapter. An debugger adapter is a core class that acts as a facade for an underlying execution engine. It is used for both accessing the control flow structures
such as contexts or variables, and for controlling the execution. Basically, there is one debugger adapter for each language.

**ContextAdapter, VariableAdapter**  
and **InstructionAdapter.** These are helper classes that provide a uniform access to the language's execution engine internals.

**DebuggerService.** A debugging service is responsible for performing debugging operations such as *step-into* or *step-over*. It is a mediator between a debugger adapter and a debugger user interface.

**Mode and subclasses.** The debugger service modes represent a debugging action to be taken next time the interpreter executes a piece of code.

**DebuggerUI.** Finally, the debugger is a user interface. It presents the source code and the execution state to the programmer. It also enables the programmer to perform debugging operations from the UI.

### 4.3.2 Debugger Adapter

As we said before, the debugger adapter is a core class of the whole system. It interfaces an underlying execution engine in general way, no matter how is it implemented. More precisely, the debugger adapter first, provides a uniform access to the current execution state, second, it emits events whenever the execution state of the program changes and
third, it provides facilities for suspending/resuming the execution. Generally, there is one debugger adapter for each interpreter. In practice, debugger adapter implementations can be shared between several languages that are implemented in the same way.

For example, the Perseus framework presented later in Chapter 6 provide a debugger adapter implementation that can be used for all the languages based on the framework\(^2\).

### 4.3.2.1 Execution State Model

The execution state modeled by a set of adapters: a context adapter, variable adapter and an instruction adapter. The adapter objects are used by the user interface to present the state to the programmer.

**Context adapter.** A context stack is model-led by the context adapters. Each context adapter belongs to one activation record on an interpreter’s execution stack. The context adapter provides an access to:

- the name of the function or method that belongs to the context,
- the source code of that function or method,
- a context adapter of the sender (caller) context (as another context adapter),
- an instruction being interpreted (as instruction adapter),
- a set of variables that belong to the context (as variable adapters).

The adapter also contains a reference to the debugger adapter it belongs to.

**Instruction adapter.** The instruction adapter represents an instruction being interpreted. It contains a line reference to the source code which is used by the debugger to visually emphasize a current position in the code. Although this object is called a instruction adapter, it is generally not related to the interpreter’s (or hardware processor’s) program counter register. Here the instruction is just an abstraction of the smallest piece of code that is executed atomically by an interpreter. An instruction can be a single bytecode or an AST node depending on the interpreter’s internal architecture.

**Variable adapter.** Variable adapters abstracts the function arguments and local variables. It also enables the debugger to read and modify variable value.

Although a presented set of adapters covers a wide range of programming languages, it does not completely cover all possible languages. New kinds of adapters and properties can be easily added by means of using customized adapters.

\(^2\)Actually the debugger adapter and language interpreter is realized by one class – the Perseus::Interpreter.
4.3.2.2 Execution Control Facilities

During an interactive debugging session, the program execution is interlaced with a debugging phase. In the debugging phase, the program execution is temporarily suspended and programmers are given a chance to interactively explore and modify program state such as variable values. At the end of the debugging phase, programmers might resume the program execution by means of a debugging operation or it can abort the execution at all.

To enable an interactive debugging, the debugger adapter exposes three methods with obvious meaning: suspend, resume and abort. Those methods are used by the debugger service to drive the execution during the interactive debugging session.

4.3.3 Debugger Service

The debugger service implements the debugging operations such as step-into, step-over and continue. It acts as a model for the debugger user interface. During a debugging session the debugger service is attached to the debugger adapter. That means that the debugger service is registered to the adapter and receives emitted announcements:

```plaintext
1 DebuggerService >> attach: anDebuggerAdapter
2   adapter ← anDebuggerAdapter.
3   subscribe: TracepointAnnouncement send: #tracepointInterrupt to: self;
4   subscribe: ContextAnnouncement send: #contextChanged to: self;
```

At the end of the debugging session, the debugger service detaches from the debugger adapter:

```plaintext
1 DebuggerService >> detach
2   adapter unsubscribe: self.
```

For more detailed description of the debugging service and debugging operation implementations, refer to [68].

4.3.4 Stacking Debugger Adapters

The basic idea of our unified debugger is following: in addition to the execution stack, a stack of debugger adapters is maintained. Each debugger adapter corresponds to a bunch of activation records on the execution stack.
A debugger adapter stack must be maintained manually, *i.e.*, a programmer should include a stack modification code into the code, whenever a program control flow enters or leave a chunk of code in other language than the one currently being executed. Two functions are provided for managing an interpreter adapter stack: `pushDebuggerAdapter:` and `popDebuggerAdapter`. When a new debugger adapter is pushed onto a stack, all debugger services attached to a current debugger adapter must attach a new debugger adapter:

```plaintext
1 DebuggerAdapter class >> pushDebuggerAdapter: newDebuggerAdapter
2   activeAdapter subscribers do:
3     [:subscriber |
4       subscriber detach.
5       subscriber attach: newDebuggerAdapter].
6   newInterpreter next: activeAdapter.
7   activeAdapter ← newDebuggerAdapter
```

Similarly, when the topmost adapter is to be removed, all attached debugging services must reattach the next one:

```plaintext
1 InterpreterAdapter class >> popDebuggerAdapter
2   activeAdapter subscribers do:
3     [:subscriber |
4       subscriber detach.
5       subscriber attach: activeAdapter next].
6   activeAdapter ← activeAdapter next
```

Manual management of the debugger adapter stack is usually not a big deal since calling foreign functions, *i.e.*, routines implemented in another language, often requires a glue code.

See the following example of multi-language applications written in Smalltalk, XQuery and JavaScript. A Smalltalk part of the application instantiates an XQuery interpreter and evaluates an XQuery code:

```plaintext
1 | xqInterpreter |
2 xqInterpreter ← XQueryInterpreter new.
3 xqInterpreter evaluate: query
```

The `query` variable holds an XQuery code defining new function for computing combinatorial numbers:

```plaintext
1 import module namespace js = "http://sma...";
2 declare function combinatorial-number ( $n , $k ) {
```
The XQuery code calls a function `js:factorial` implemented in JavaScript:

```javascript
function factorial ( a ) {
  if ( a == 0 ) {
    return 1;
  } else {
    return a * factorial ( a );
  }
}
```

Figure 4.2 shows an execution and debugger adapter stack for the example above.

The method `evaluate:` in a unified-debugger-enabled XQueryInterpreter class is, in principle, implemented as follows:

```javascript
XQueryInterpreter>> evaluate: query
| queryTree result |
queryTree ← self parse: query.
DebuggerAdapter pushDebuggerAdapter:
  (XQueryDebuggerAdapter on: self).
result ← self visit: queryTree.
```
The XQuery interpreter supports a number of primitives. Primitives are the functions that are directly callable from an XQuery code, but whose implementation is done in another language than XQuery. The XQueryInterpreter calls the method `performJsPrimitive:withArguments:` to call the primitive implemented in JavaScript:

```
XQueryInterpreter >> evaluateJsPrimitive: primName withArguments: args
| result |
DebuggerAdapter pushDebuggerAdapter:
  (ByteCodeDebuggerAdapter on: self).
result ← jsPrimitiveLibrary
perform: primName withArguments: args.
DebuggerAdapter popDebuggerAdapter.
↑ result.
```

Figure 4.3 shows a user interface for our unified debugger implementation. A user can explore the stack and resume the evaluation in both step-into and step-over manner.
4.4 Related Work

NetBeans and Eclipse are integrated development environments (IDEs) used mostly for the development of Java applications. Since they have been designed with respect to general-purpose use, the plugins for other languages (e.g. Groovy or Ruby) are available.

It seems that the architecture of the debugger subsystems in both IDEs had two main design goals: to enable smooth pluggability of new debuggers and to facilitate a code reuse. In our terminology, their architecture is composed of a debugger adapter sending events directly to the user interface. An user interface code is the same for all debugger adapters, which means once a debugger for a given language was developed it is relatively easy to write bindings that will enable its usage in NetBeans or Eclipse.

NetBeans IDE is capable of debugging multiple-language applications as long as all the debugged languages are compiled into the Java bytecode. It uses class file debug symbols that map individual bytecode instructions to tuple (file name, line number). In other words, the NetBeans debugger makes use of a standard JPDA [4] interface not knowing whether the (byte)code currently under execution has been compiled from Java, Groovy, Ruby or Python.

This approach has one advantage: a new language running on the top of the JVM can be debugged immediately (assuming its compiler generated correct debug symbols). On the other hand, this approach implies the debugger to be aware only of the JVM metamodel and it can not accurately display the information that goes beyond this metamodel—for example, it is completely oblivious to Python’s dynamically added instance variables. For this reason, we consider such an approach not scalable enough in the long term.

4.5 Summary

The debugging of multiple-language applications is difficult and current debuggers offer little help to alleviate the problem. The only debuggable multiple-language applications are those composed of languages compiled to the same runtime, e.g., applications composed of C and C++ code, applications composed of Java, JRuby and Jython code etc.

As a remedy, this section proposed an architecture that is able to merge multiple single-language debuggers into a unified one capable of debugging multiple-language applications. The architecture introduces a new abstraction layer—the debugger adapter—which interfaces with underlying execution runtimes. The stack of debugger adapters then manages the transition of control from one debugger adapter to another.

The validity of this architecture is based on two assumptions: first, it is always possible to intercept a message invocation and second, it is always possible to tell the language of the method that is about to be invoked. The first assumption is always fulfilled, as all real-world languages do have debuggers capable of emitting the event on a method dispatch. The second assumption is fulfillable easily: compilers of each language used in
the application only need to generate a debug symbol (or an annotation) denoting the language of the currently compiled code.

We believe that the architecture is easily extensible to merge the debug information from the debuggers running in different processes. That makes it a good candidate for debugging RPC-based applications as well as systems in which method invocation results in a new process creation (such as shell scripts). In the future, we also plan to integrate the unified debugger into an Smalltalk/X environment.
5 VM-level MOP for the Method Lookup

5.1 Introduction

In modern object oriented programs the message send (method call) represents a vast majority of operations performed. Message send is also a time-consuming operation because for each message send, a corresponding method has to be found. The method lookup usually involves searching through multiple method dictionaries over the class inheritance hierarchy. The whole logic of the message send, including the method lookup algorithm is usually implemented at the VM level and exposed by means of “send” or “invoke” primitive operations of an intermediate language (a bytecode in most cases). This approach has two advantages: it keeps a program representation small and, what is more important, gives the VM chance to optimize a method lookup. Various techniques have been developed to speed up the lookup or avoiding it at all, while preserving original semantics.

As we have shown in Section 2, the method lookup semantics differs from language to language. The lookup performed by the VM matches the lookup of the language the VM has been designed for. When implementing a new language on the top of the existing VM, the language implementers have basically two options how to deal with an incompatible lookup algorithm: either to modify the VM to suit their needs better or to simulate required behavior on the object-level. We think that changing the VM is not the way to go. Changing the VM is difficult, time consuming task, especially for highly optimized VMs. This approach also does not scale well. What happens, for example, if different parts of a code running within a single VM need different lookup algorithms? Finally, sometimes it is not possible to change the VM at all.

On the other hand, using the object-level simulation APIs usually implies a performance loss in the order of magnitude. Therefore mechanism how to hookup a custom method lookup is highly desirable.

In this section, we propose a simple virtual machine-level metaobject protocol (MOP) for controlling the method lookup, which is generic and which does not imply a noticeable runtime overhead. We also show few case studies where the MOP helps to implement new semantic requirements. Although the protocol has been designed and implemented for a Smalltalk virtual machine, the same approach can be used for any class-based language environment.

This chapter is organized as follows: Section 5.2 formulates requirements for the MOP, Section 5.3 describes the MOP we have designed. Section 5.4 presents some examples of using the MOP and validates its design. Section 5.5 discusses some implementations issues, performance and memory consumption based on benchmarks. Finally, the Section 5.7 discusses the ways of the future research.

\footnote{For example, sources of Microsoft’s CLR or Cincon’s VisualWorks are not publicly available.}
5.2 Requirements

As we said before, a VM-level method lookup implementation provides reasonable performance but poor flexibility. An object-level lookup implementation is flexible enough but lacks the performance. We want to make the best of these two worlds – a metaobject protocol:

**Simple.** The MOP should be generic but simple. It means that it is both simple to implement and simple to use.

**Backward compatible.** This requirement comes from the practice. Lots of codes have been already written and well tested. Such codes should run unmodified on the MOP-enabled VMs.

**No performance overhead.** As we have shown before, an alternative solutions based on object-level code usually lacks adequate performance. Mature virtual machines use various techniques to speed up a method lookup – inline caches, polymorphic inline caches and method inlining in particular. The MOP should smoothly cooperate with them.

5.3 Method Lookup Metaobject Protocol

The basic idea of the method lookup MOP is simple: to provide a mechanism to hook up a user function that will be invoked by the VM, whenever the method must be looked up. In object oriented languages, a widely used idiom to realize a callback is to provide an object that implements a method with a predefined signature. The proposed protocol uses the same approach: the VM invokes a *lookup method* on a *lookup object*, passing all required information as parameters. The returned method is then executed as a response to the particular message send.

The MOP-enabled method lookup is split into two phases:

- **Lookup object lookup.** Within a lookup phase, a sending method and receiver’s class is searched for a *lookup object*. The lookup object is responsible for looking up the method for a given method name (selector).

- **Lookup.** Once the lookup object is determined (by the *lookup object lookup*), the VM asks it for a method, passing a method name, receiver’s class and other information required for the method lookup. The result of the lookup will be either a method to execute or nil, which means that the message is not understood by the receiver.
5.3.1 Lookup Object Lookup Phase

The lookup object is responsible for looking up a method, given selector and receiver’s class. In most modern object-oriented languages the selection of an actual method to be executed depends solely on the method selector and receiver’s class. Such behavior is considered one of the most fundamental principles of an object-oriented programming paradigm. However in certain cases the method choice can depend also on the calling context. As described in the previous section, in classbox-enabled environments, the code to be executed also depends on the classbox of the originating method. The same holds for other module systems, e.g., selector namespaces [69, 60] or changeboxes [72].

The lookup object lookup algorithm is shown in Figure 5.1. The first step of the lookup object lookup process (lines 4-5) gives us the possibility to override the method lookup on a per-sending method basis. Searching for lookup object within the class hierarchy (lines 6-11) gives us a possibility to define the lookup objects at one place without having to deal with subclasses, which may be dynamically loaded or created later.

The lookup object provided by the sending method always takes precedence to the lookup provided by the receiver’s class. Since the lookup objects are accessible by the reflection API, a method-specified lookup object can communicate with receiver’s class lookup object.

```ruby
1 Object lookup_lookup_builtin ( Method sending_method,
2   Class receiver_class )
3 {
4   if (sending_method.lookup != nil)
5     return sending_method.lookup
6   Class current_class = receiver_class;
7   while (current_class != nil) {
8     if (current_class.lookup != nil)
9       return current_class.lookup;
10    current_class = current_class.superclass;
11  }
12  return nil;
13 }
```

Figure 5.1: Lookup object lookup algorithm

5.3.2 Lookup phase

The metaobject protocol-aware method lookup routine is shown in Figure 5.2. Whenever the method must be looked up, the virtual machine uses this routine to determine which method should be executed.
Method method_lookup (String selector, 
   Class search_class, Method sending_method )
{
   Object lookup;
   Method method;
   lookup = lookup_lookup_builtin(sending_method, 
      receiver_class );
   if (lookup == nil && check_not_recursive(lookup) {
      method = lookup_lookup_builtin(selector, receiver_class, 
         sending_method);
   } else {
      push_lookup(lookup);
      method = send(lookup,"lookup", selector, 
         receiver_class, sending_method);
      pop_lookup();
   } 
   return method;
}

Figure 5.2: Method lookup routine hard-coded to the virtual machine

There is one place, where the virtual machine dispatches back to the user-level code: when a custom lookup object is provided (see Figure 5.2, line 13). Since the invocation of lookup routine of lookup object is implemented by a normal message send, the lookup may lead into the endless recursive invocation of the lookup routine. To catch such a situation, a thread-local stack of activated lookup objects is maintained (lines 13 and 15). If a recursive activation is detected (by examining the lookup object stack), a builtin lookup is used. This mechanism enables setting a lookup object on the root of a class hierarchy and, to some extent, it makes possible to use the customized lookup even for the lookup objects.

5.3.3 Custom Lookup Routine

Since the results of the method lookup are subject to memoization, a user-provided lookup routine is required to be effect free. Such property is not, however, enforced and it hardly could be. On the other hand, a user code should manually trigger flushing all involved caches when necessary. For example, setting a custom lookup object for a class should result in a cache flush since all cached lookups for a given class or subclasses might be invalid since then;
5.4 Validation

We have successfully implemented the metaobject protocol as described above in the Smalltalk/X virtual machine \(^2\) [5]. In this section, we will discuss few examples of the MOP usage, which will validate the implementation.

5.4.1 Method Privacy

Many object oriented languages provide a mechanism to limit the access to the object’s methods. A method can be marked either public (any method can invoke such a method), protected (only the methods that belong to same class hierarchy as the protected method can invoke it) or private (only the methods from same class can invoke the private method). Traditionally, a Smalltalk language enforces all methods to be public.

A trivial implementation of method privacy on the top of method lookup MOP can be:

```plaintext
PrivacyLookup >> lookupMethodForSelector: selector directedTo: searchClass
   from: sendingMethod

| method |

method ← StandardLookup new
lookupMethodForSelector: selector directedTo: searchClass
   from: sendingMethod.
method isPrivate ifTrue:
   [ ↑ (method containingClass == sendingMethod containingClass)
      ifTrue: [ method ]
      ifFalse: [ nil ] ].
method isProtected ifTrue:
   [ ↑ (method containingClass
      sameHierarchyAs: sendingMethod containingClass)
      ifTrue: [ method ]
      ifFalse: [ nil ] ].
↑ method.
```

5.4.2 Selector Namespaces

As we said in Section 2.1.3 the selector namespaces provides a powerful tool to cope with the software evolution and composition. We have successfully the implemented selector namespaces on the top of the MOP described above. A selector namespace support has already been available for testing as of Smalltalk/X 6.1.1β. For more detailed description

\(^2\)A MOP-enabled VM is available at https://swing.fit.cvut.cz/projects/stx-goodies/wiki/Download
of the implementation of the selector namespaces in Smalltalk/X environment refer to Chapter 7.

5.4.3 SmallRuby

SmallRuby [A.13] is another implementation of the Ruby programming language built on the top of the Smalltalk/X virtual machine. Although it is said that Ruby is a reborn of Smalltalk, its lookup algorithm differs from the Smalltalk’s and thus the Smalltalk native lookup algorithm is hardly usable to cover full semantics of Ruby. The coexistence of Smalltalk and Ruby in single environment poses other complications.

SmallRuby exploits the method lookup MOP to full implementation of the Ruby’s semantics. For more details about SmallRuby and its implementation, see Chapter 8 and Section 8.3, respectively.

5.5 Discussion

5.5.1 Cache Effects

Most VM implementions use various caching strategies to reduce or eliminate performance penalties of the dynamic method lookup algorithm. Smalltalk/X uses a three-layer cache hierarchy consisting of first, an inline cache (ILC), second, a polymorphic inline cache (PIC) and third, a global selector cache. Inline caches remember the code-pointer of the last lookup on the call site, as described in [25] and [66]. These caches reduce the message-send overhead to a single direct or indirect function call instruction, followed by a receiver-class check in the called method (a compare and branch). For a hit, i.e., the send is to an instance of the same class as cached, the dynamic overhead consists of a function call followed by a compare and an untaken conditional branch. For the miss, a polymorphic inline cache is consulted, which provides code addresses for a small (< 20) number of receiver classes organized as LRU of the recently called targets [35]. Finally, if the PIC also fails providing a target address, the VM’s builtin lookup method is invoked, which performs a full search, fills PIC and/or ILC cache slots for the next call, and passes the control to the target method. This builtin full lookup uses a global selector cache, but its effect is actually marginal.

The MOP as described is invoked before the builtin full lookup is consulted - actually, it is sliced inbetween the PICs and the full lookup. However, its result (if non-nil) is placed into the ILC or PIC, as usual. Therefore, the performance penalties of the MOP, as compared to the builtin lookup, are seen only in cache miss situations. As in typical programs, ILC hit rates are above 95% and PIC hits are typically above 90% of the remaining 5%, the relatively slow MOP performance is hardly noticed, except for a very polymorphic code, e.g., when iterating over all objects in the system or when very deep inheritance hierarchies are used, which overflows the configured PIC sizes [31].
5.5.2 Method Inlining

ILCs and PICs significantly improve performance of message sends by eliminating the need for full method lookup in most cases. Advanced compilers go beyond that and generate even more optimized code by inlining target method into the code of the sending method, effectively eliminating the overhead connected with the execution of the method. The proposed MOP does not impose any difficulties for inlining compilers. The inlined method must be looked up anyway thus the only thing to change is to update compiler routine to consider the MOP. In most cases the inlining compiler exploits the data in ILCs and PICs.

5.5.3 Multiple/Predicate Dispatch

The multiple dispatch and predicate dispatch are very interesting lookup algorithms. MOP as described above provides no direct support for these method lookup algorithms because they would require more complex caching than the one described earlier in this section. Basically, there are three options how to implement a multiple or predicate dispatch: first, dynamically compile object-level code emulating multiple/predicate dispatch, second, extend MOP to pass also all arguments and turn off all caching mechanisms (which will have drastic effect on performance) or third, generalize caching strategies to cache target method on any of arguments using generic, user-provided predicate. However, the last option would require complex changes to the VM and it is now left as future work. The design of the proposed MOP is a compromise between its generality and simplicity (in terms of the implementation).

5.5.4 Performance and Memory Overhead

The detailed analysis of performance benchmarks and memory overhead of the MOP implemented in Smalltalk/X VM is presented in Section 7.2.2.1.

5.6 Related Work

5.6.1 Smalltalk #doesNotUnderstand:

In Smalltalk, when a message is not understood by the receiver, the VM sends a #doesNotUnderstand: message to the original receiver along with original arguments and selector. The return value of #doesNotUnderstand: method is returned to the sender of the original method. By a custom implementation of #doesNotUnderstand: method, programmers can define their own behavior and semantics or message sends. Use of #doesNotUnderstand: mechanism is slow. Results of (failing) lookup are not cached, therefore every invocation of #doesNotUnderstand: is preceded by a (slow) full method lookup. Moreover, the arguments to the original message together with the selector need to be packed into a new object in
order to be passed to \#doesNotUnderstand: method. Custom method dispatch semantics implemented by \#doesNotUnderstand: mechanism is roughly thousand times slower than the lookup performed by the VM. Since the MOP proposed in this chapter is implemented at the VM level, it provides the flexibility comparable to \#doesNotUnderstand: and performance comparable to builtin lookup as performed by the VM.

5.6.2 JSR 292: Invoke Dynamic

JSR 292: Invoke Dynamic [37] introduces a new general mechanism for method calls into the Java virtual machine. A new bytecode instruction – the invokeword – with few supporting objects allows language implementers to intercept method calls at the call site and provide their own method lookup logic [54]. The main motivation for introducing invokeword was to better support dynamic languages in the JVM.

The purpose and usage of the described metaobject protocol and JSR 292 are similar. Both of them could be used to provide user-specific method lookup algorithms without performance loss. In both cases, the user must explicitly mark objects with a user-defined lookup and the lookup itself can be implemented as normal Smalltalk or Java code without any need to change the virtual machine itself. However, our metaobject protocol differs from JSR 292 in two important points: first, the JSR 292 introduces a new instruction whereas our metaobject protocol does not and second, the JSR 292 allows method lookup interception only at the call site whereas our protocol allows both, on the call site and (possibly at the same time) at the receiver site.

These limitations make invokeword difficult to use in certain cases. The use of a new instruction means that it is impossible to alter the method call semantics externally, without recompiling or instrumenting the code. Lack of possibility to intercept method lookup at the receiver site makes it hard to use (if even possible) in cases where we want to alter semantics regardless of the call site. For example, invokeword would be hardly used for the implementation of mixins or traits [56].

5.6.3 .NET DLR

Dynamic Language Runtime [23] (DLR) is .NET library that eases the implementation of dynamic languages for the .NET platform. To implement custom method lookup semantics, DLR provides infrastructure similar to that provided by Java and JSR 232 (invokeword). At the moment, DLR is implemented purely at the object level and does not have any support in the virtual machine. Therefore programming languages implemented using DLR have the same problem as described those earlier in Section 2.2.2: code written in dynamic language is much slower than equivalent code written in C#. Contrary to DLR, a proposed MOP is implemented at the VM level that’s why the code using the MOP for a custom method lookup executes nearly at the same speed as the code written in Smalltalk (in our case).
5.6.4 COLA

In COLA based languages [52], an object behavior is defined by its v-table object. A method is looked up by sending a “lookup” message to the v-table passing the selector as an argument. A programmer may alter a message lookup algorithm by redefining or overriding a lookup method in object’s v-table. The described MOP differs from the COLA’s approach. The described MOP passes a method selector, search class, and sending method as arguments, whereas in a COLA system the method selector is the only arguments it pass.

5.6.5 JikesRVM

Modular architecture of a JikesRVM virtual machine [38] is open to customization, including customization of method lookup algorithm. However, to alter a method lookup algorithm in JikesRVM, the VM must be modified, rebuild and distributed along with the program using the custom lookup – contrary to our MOP. It also provides no facilities to have different lookups for different classes, whereas this is possible by using the proposed MOP.

5.7 Summary

In this chapter we claim for the need for a virtual machine level metaobject protocol for customizing method lookup and we presented such a metaobject protocol. We have validated the protocol by implementing two extensions to the Smalltalk language. We have also shown that the possibility to controlling the method lookup is very useful for porting existing languages to a new platform.

We plan to extend the metaobject protocol at the object level and provide mechanisms to combine multiple custom lookups that are not aware of each other. Another interesting direction is to explore how to decouple the VM from the class objects and their internal layout, which now must be known to the VM.
Part II

Validation
6 The Perseus Framework

Perseus [A.15] is a framework built on top of debuggable interpreter design pattern that eases the construction of programming environments. It has been written by Alexandre Bergel and the author of this thesis. For a given grammar and a set of abstract syntax tree nodes following a reduced set of idioms described above, Perseus provides a programming environment that includes a debugger and an event mechanism. This mechanism, called announcement, may be used to build execution analyzers, such as tracers and profilers. The Perseus framework is freely available for both Smalltalk/X and Squeak.

6.1 Using Perseus for Language Construction

This section describes a typical instantiation of the Perseus framework.

For an end-user, the creation of a programming environment requires two distinct steps: constructing the abstract syntax tree and assembling the interpreter.

6.1.1 Abstract Syntax Tree Construction

Abstract syntax tree nodes are instantiated during the parsing phase by a parser that may be produced by SmaCC [17]. Following the traditional approach to build a parser, an excerpt of the SmallScript grammar is (using the SmaCC syntax):

```plaintext
... if_statement: "if" "(" expression ")" statement "else" statement
 { SmallScript::IfThenElseNode new
   condition: '3';
   then: '5';
   else: '7' } ;
...
function_call: id "(" arguments ")"
 { Perseus::FunctionCallNode new
   name: '1' value;
   arguments: '3';
   yourself } ;
...
```

Both classes SmallScript::IfThenElseNode and Perseus::FunctionCallNode define a method acceptVisitor: that simply dispatch this call to a visitor.

Perseus provides a set of generic nodes that can be used when defining the AST construction for a new language. Perseus::FunctionCallNode is part of Perseus, whereas

1This example is based on Smalltalk/X version of the Perseus
SmallScriptIfThenElseNode must be defined. All parse nodes must be a descendant of the Perseus::ParseNode.

6.1.2 Writing an interpreter

SmallScript::Interpreter implements the visiting methods that performs actual interpretation. This class must be a subclass of another Perseus::Interpreter.

```smalltalk
Perseus::Interpreter subclass: #SmallScript::Interpreter
instanceVariableNames: ''
```

The visiting methods corresponding to the function call and conditional nodes are:

```smalltalk
SmallScript::Interpreter>> visitSmallScriptIfThenElseNode: aNode
  condition ← self visit: aNode conditionNode.
  ↑ condition
  ifTrue: [ self visit: aNode thenNode]
  ifFalse: [ self visit: aNode elseNode]

SmallScript::Interpreter >> visitPerseusFunctionCallNode: aNode
  |
  name args |
  name ← aNode name.
  args ← aNode arguments collect: [:arg | self visit: arg].
  ↑ self evaluateFunction: name withArguments: args

SmallScript::Interpreter >> evaluateFunction: name withArguments: arguments
  |
  currentContext functionContext functionNode returnValue |
  "Retrieve the function node"
  functionNode ← self functionAt: name.
  "The context has to be set back before exiting this method"
  currentContext ← context. "context is an instance variable"
  "functionContext is a new context used to evaluate the function"
  "Evaluated arguments are associated to names in functionContext"
  functionNode argumentNames with: arguments
  do: [:nameNode :value |
    functionContext
    at: nameNode name put: value].
  context ← functionContext.
  self announceFunctionEntered.
  "Evaluate the function body"
```
These methods are invoked when a SmallScript program is interpreted. They follow the three idioms set by PERSEUS:

1. the recursion over the tree must be done through the visit: method,
2. each node that can contain a breakpoint should invoke announceTracepoint-Interrupt,
3. function entering and exiting must be signalled with announceFunctionEntered and announceFunctionLeft.

### 6.2 Graphical Tools Provided by PERSEUS

The PERSEUS framework provides an extensive set of graphical tools for evaluating and debugging programs written in PERSEUS-based programming languages. A graphical user interface is accessible through the class Perseus::WorkspaceApplication as in the following example:

```ruby
Perseus::WorkspaceApplication openFor: SmallScript::Interpreter new
```

Figure 6.1 shows a workspace window for the SmallScript language. A menu is accessible through a right-click enabling breakpoints to be set and removed. Like in Smalltalk workspace, the selected program is executed by choosing Do it item from context menu, the Print it and Inspect it evaluates the code and displays the result. Finally, the Debug it runs the selected code in a visual debugger (Figure 3.2). The operations continue, step-over, step-into and terminate can be invoked.

### 6.3 Writing Tools on the Top of the PERSEUS Framework

The programming environment offered by PERSEUS may be enhanced in several different ways. We will successively see the implementation of the conditional breakpoints, the use of announcements for dynamic analysis, and new debugging facilities that operate over the program history.
6.3.1 Conditional Breakpoints

A breakpoint in the program can be set by designing a location in the source code and then choosing the ‘add breakpoint’ entry in the menu offered by the workspace. The AST node corresponding to a location in the source code is trivially obtained from a visitor that simply checks for each node if the location is comprised in the syntactical delimitation of the node. The only requirement for this is that each node must know its delimitation in the source code. As mentioned earlier, a service contains the list of breakpoints. In a continue mode, the presence of the breakpoint is checked for each node (Continue >> tracepointInterrupt presented in Section 3.3.6). The way the breakpoints are managed enables new kinds of breakpoints to be easily defined. A subclass of Breakpoint can be defined as follows:

1. Breakpoint subclass: #ConditionalBreakpoint
2. instanceVariableNames: ’condition’

ConditionalBreakpoint is a breakpoint that halts the program execution if a condition is fulfilled:

1. ConditionalBreakpoint >> isBreakpointOn: aNode interpreter: anInterpreter
SECTION 6. THE PERSEUS FRAMEWORK

Having breakpoints as first class entities is a key point in Perseus. In a similar fashion, new kinds of breakpoints can be easily added that may be related to the context modification or to a particular control flow.

6.3.2 Dynamic Analysis

In Section 3.3 we presented the announcement mechanism and the way it may be used to define graphical user interfaces. Announcements enable the creation of dynamic analyzing tools. For example, an elementary program tracer might be:

```
InterpreterObserver subclass: #SimpleTracer
    instanceVariableNames: '

SimpleTracer >> registerAnnouncementHandlers
    super registerAnnouncementHandlers.

SimplerTracer >> onAnnouncement: anAnnouncement
    Transcript show: anAnnouncement printString; cr
```

The message `observe: Announcement send: #onAnnouncement: to: self` registers the tracer as a listener to any announcement. Whenever an announcement is emitted, the method `onAnnouncement:` is invoked. The following code makes a tracer listen to events generated by the service:

```
SimpleTracer new debuggerService: Perseus::DebuggerService
```

An automatic invocation of `registerAnnouncementHandlers` insures a proper registration. An excerpt of the output generated by the execution of the factorial example is:

```
<interpreter starts>
<SsmallScriptAssignment>
'a' <- 6
```
As further example we can present a profiler that assesses the amount of time spent in functions. The class `SimpleProfiler` contains two instance variables:

```smalltalk
InterpreterObserver subclass: #SimpleProfiler
instanceVariableNames:
'functionInvocationCounts functionAverageNodesExecuted'
```

The variable `functionInvocationCounts` refers to the number of invocations for each function of the interpreted program, the number of times it has been invoked. `functionAverageNodesExecuted` refers to the “length” of each function in terms of nodes that have been traversed. These variables are initialized with an empty dictionary when `SimpleProfiler` is instantiated.

```smalltalk
SimpleProfiler >> initialize
functionInvocationCounts ← Dictionary new.
functionAverageNodesExecuted ← Dictionary new
```

When a function is entered, its corresponding number of invocations is incremented by one. For each traversed node, the function in which this node is contained, is incremented by one.

```smalltalk
"Function entering"
SimpleProfiler >> onFunctionEntered
| function |
function ← interpreter context function
functionInvocationCounts at: function put: (functionInvocationCounts at: function ifAbsent: [0]) + 1)

"Beginning of the interpretation"
SimpleProfiler >> onInterpreterStarts
functionInvocationCounts at: interpreter context function put: 1

"Node traversed"
SimpleProfiler >> onTracepoint
functionAverageNodesExecuted at: interpreter context function put: (functionAverageNodesExecuted at: interpreter context function ifAbsent: [0]) + 1
Some printing functions may be necessary to display the results of the profiling:

```smalltalk
defineMethod: SimpleProfiler printStatisticsOn: aStream
  totalNodes ← 0.
defineMethod: SimpleProfiler functionAverageNodesExecuted:
  functionInvocationCounts keysAndValuesDo:
    function ← function at: :function.
    nodesPerFunction ← functionAverageNodesExecuted at: function.
    average ← (nodesPerFunction / totalNodes) asFloat.
    totalNodes ← totalNodes + count.
```

A typical output looks like:

```
PSSimpleProfiler statistics:
f1 55.55555555556% 2 5.0
f2 38.8888888889% 2 3.5
main 5.555555555555% 1 1.0
```

In terms of the number of the nodes traversed, 55% of them are the part of the function f1, 38% of f2 and 5% of the entry point.
6.4 Practical Applications of PERSEUS

The PERSEUS framework is currently successfully used in several other projects and applications.

6.4.1 CellStore/XQuery

CellStore/XQuery is an Smalltalk implementation of the XQuery language [15] developed within the CellStore project [A.14, 71]. An XQuery interpreter is implemented using PERSEUS framework and therefore comes up with wide set of tools including visual debugger.

6.4.2 IZAR

IZAR is a software tool for both single-criteria and multi-criteria decision making [A.16, 40, 41]. IZAR supports several algorithms and provides unified, convenient user interface. One of the primary goals of IZAR was to make the application extensible i.e., to provide mechanisms to modify the existing algorithms for solving multicriteria problems or to add a completely new one. The second goal was to use rational arithmetic for all computations. The whole application is written in Smalltalk/X. However, the lack of certain language features such as a direct multi-dimensional array indexing and the compactness of algorithms makes it difficult to express in a Smalltalk programming language. Moreover, end-users and algorithm developers (mostly mathematicians and economists) understands Algol-like languages such as C, Fortran and Pascal rather than Smalltalk.

In IZAR, decision-making algorithms (representing more than half of the IZAR code) are implemented in a Pascal dialect. Its interpreter is based on PERSEUS and PERSEUS-based tools such as debugger and tracer have been used for the developing, debugging and profiling decision-making algorithms.

6.5 Summary

In this section, we presented the PERSEUS – a framework for building programming language interpreters. PERSEUS has been based on the debuggable interpreter design pattern described earlier in Chapter 3. Our visual source-code level debugger, which is also a part of PERSEUS, may be used to test and debug code executed by any PERSEUS-based interpreter. We have also shown how to add new features to PERSEUS.
7 Multi-language Support in Smalltalk/X

The first part of this thesis provides a description of concepts and approaches we use to build a multi-language programming environment. We believe, however, that the ideas must be evaluated also on large scale, real-world applications. We believe that feedback based on every-day usage is really important. Therefore, we put a significant effort into developing and maintaining a version of Smalltalk/X that adopts some concepts described above.

In this chapter, we will briefly describe improvements we have made to Smalltalk/X platform and the development environment. This modified Smalltalk/X environment – referred as “jv branch” [A.12] – is publicly available in the form of ready-to-use, precompiled software packages for both Linux and Windows operating systems. The source code of a modified class library is available as well.

Some of the modifications we did to the Smalltalk/X class library and to the VM has been already integrated to the main version of Smalltalk/X, and therefore used by the developers all over the world. Other improvements will be integrated in a near future.

All these changes have been designed with one important aspect: backward compatibility. Since Smalltalk/X is heavily used in the industry, backward compatibility has to be maintained at least at the source code level.

7.1 Brief Overview of Smalltalk/X

Smalltalk/X is a complete reimplementation of Smalltalk-80. Its development was started in mid 80s by Claus Gittinger. In these days the development is done by eXept Software AG. Originally, Smalltalk/X was purely batch-oriented system like most of the traditional programming environments. A Smalltalk source code was prepared in the text editor, then translated by a specialized compiler called \texttt{stc} into a equivalent C source. C code was in turn compiled by a C compiler into a binary executable for the given platform. The bytecode interpreter, incremental compiler and just-in-time compiler was added later covering the need for more agile development environment.

Nowadays Smalltalk/X features are comparable to classical Smalltalks like Squeak [36] or VisualWorks [67]: it provides fast optimized VM, rich class library including a user interface framework, modern integrated development environment and the support for an image-based style of development. In addition, the Smalltalk/X still provides the \texttt{stc} compiler allowing developers to deploy Smalltalk applications in the form of a set of shared libraries and binary executable. It also maintains a binary compatibility between the \texttt{stc}-compiled code and the code compiled by the incremental bytecode compiler. In fact, the \texttt{stc} compiler is still necessary to bootstrap the Smalltalk/X environment from the source code.

On the other hand, the existence and necessity of the \texttt{stc} compiler makes any change to
the VM and language even more difficult to make because any change in the VM implies an appropriate change in the stc compiler.

7.2 Method Lookup MOP

We have successfully implemented the method lookup MOP from Chapter 5 in Smalltalk/X VM. The implementation has been done partially by Claus Gittinger, especially the parts involving the context management and the just-in-time compiler.

7.2.1 Differences

The actual implementation of the MOP in Smalltalk/X slightly differs from the one described in Chapter 5: it passes more arguments. More precisely, the signature of the message that VM sends to a custom lookup object is:

```
lookupMethodForSelector:selector
directedTo:class for:receiver
withArguments:args from:context
```

The selector and class arguments are the same as in the MOP described earlier. MOP implementation in Smalltalk/X also passes a receiver object (not only its class) and the array of actual arguments (args). These arguments allow experimenting with some more complex lookup algorithms like multiple and predicate dispatches. However the VM still caches the results of the method lookup based on the receiver classes only, thus these arguments are nowadays practically useless. A generalization of ILCs and PICs to efficiently support method lookup algorithms based on generic predicates on the receiver and the arguments would be certainly interesting feature and we left this as a future work.

The context parameter represents an activation record of the sending method i.e., the method which contains a message send that caused the method lookup. The sending method as passed in original MOP described in the previous chapter can be obtained by sending a message #method to the context.

7.2.2 Performance and Memory Overhead

7.2.2.1 Performance Overhead

A natural question to raise concerns is about the actual performance overhead of the MOP. To measure an actual overhead, a set of benchmarking programs was run on Smalltalk/X for four different configurations:

STD VM - a standard Smalltalk/X virtual machine without a MOP support. A lookup algorithm is hard-coded in the virtual machine.
MOP VM, no lookup object – MOP enabled VM with no lookup object set for any class. However, the VM must check for lookup objects before doing a builtin lookup.

MOP VM, std. lookup object (C optimized) – MOP enabled VM with user-defined lookup object set on class `Object` (which is the worst case as the custom lookup is performed unconditionally for every message send to any object). The custom lookup object implements a standard Smalltalk lookup. The actual implementation is optimized at the C level and uses a global selector cache.

MOP VM, std. lookup object (pure Smalltalk) – this represents the worst case. A MOP-enabled virtual machine with a custom lookup object set for class `Object`. The actual implementation is written purely in Smalltalk and does not use any caching.

Benchmarking programs consist of both micro and macro benchmarks:

- Unimorphic sends – 1 000 000 message sends, all send the same message to an instance of the same class.
- Polymorphic sends \(x\) – 1 000 000 message sends, all send the same message each time to an instance of a different class. \(x\) stands for a total number of unique classes. Tests were run for \(x \in \{2, 4, 20, 512\}\).
- Web Application – a real application benchmark. It measures time required to process a HTTP request made to a simple web application written in the AidaWeb framework running on the Swazoo HTTP server (both are written purely in Smalltalk).
- File browser – a real application benchmark that opens a visual file browser and displays the content of a text file.
- Class Browser – a real application benchmark. It opens a standard Smalltalk/X’s class browser, switches to a given method and opens a new browser view with all references to that method.

Result analysis. Figure 7.1 shows the slowdown ratio for each benchmark when compared to the results obtained by running the benchmark on a standard unmodified VM. For the complete experiment data refer appendix A.

Unimorphic and polymorphic sends up to 20 classes. The results show the MOP does not impose any noticeable runtime overhead. This is due to the use of ILCs and PICs – the method is looked only for the very first message send for each unique class. In 1 000 000 sends, the overhead of even pure Smalltalk lookup method is hardly noticeable. The time variations are within the precision of the time measurement.

Polymorphic sends 512. This scenario represents a super-polymorphic code (512 different classes) which overflows the capacity of PIC (which is 21 in case of Smalltalk/X). Therefore,
the PIC fails each time and the full method lookup is performed. Since the class hierarchy is flat and the method is always found in the receiver’s class, the overhead is caused just by dispatching back to the Smalltalk code, which involves slow context manipulation.

**Web Application.** A web Application framework and HTTP server represents a typical application. Vast majority of polymorphic message sends fits into a PIC and thus the runtime overhead of an MOP lookup is negligible.

**File Browser.** Opening a file browser in a twice as slower on a MOP-enabled VM as on a standard VM. The reason is that the user interface core results in a highly super-polymorphic code. The user interface (UI) event dispatcher needs to iterate over all widgets composing the window. In case of file browser, there are some 30 different widget classes used, which overflows PIC capacity. Therefore, PIC fails each time an event is to be processed. We can also see that the lookup object lookup introduces a significant overhead in case of deep class hierarchies, e.g., a widget class hierarchy.

**Class Browser.** Opening class browser is approximately 14 times slower in the worst case. There are two reasons: first, it is a user-interface code and therefore suffers from the same problem as in a previous case. Second, its logic need to iterate over all classes in the system several times (to gather classes names, icons etc. – detailed analysis shows that over 3 000 000 of full lookups need to be performed just to open a class browser window). Obviously this is a highly polymorphic code and therefore PICs does not help.

The computer used for this experiment is Intel Core 2 Duo 2.00GHz, 3GB RAM, running Linux 2.6.31 and Smalltalk/X 6.1.1β.

**Summary.** The benchmarks show that the overhead introduced by the MOP is hardly noticeable in cases, where ILCs or PICs are used. This property holds for vast majority of the code. Based on the detailed analysis, the runtime overhead in UI applications is caused mostly by the UI framework and its actual implementation in Smalltalk/X.

The overhead caused by traversing a class hierarchy in lookup object lookup can be easily avoided by using a global cache mapping class object to its lookup object.

There is no well-known solution how to speed up a super-polymorphic code. It is said that such a code is very rare (less than 5% of the whole program), however the overall slowdown of UI tools is evident. One solution might be to implement a negative cache proposed by Claus Gittinger. The idea is to detect a super-polymorphic send site and then remember the code and “exception classes”. If the class of a given receiver is one of the exceptional classes, a full lookup will be performed. Otherwise, a cached method will be executed. Nevertheless, such a cache has not been ever implemented. Experiments with a negative cache was postponed to the future work.

### 7.2.2.2 Memory Overhead

The implementation of the method lookup MOP inevitably brings a memory overhead by means of memory consumption. Each class and each method in the running system must
Figure 7.1: Performance benchmarks of an MOP-enabled VM
contain a reference to a lookup object, even if no custom lookup object is set.

A basic running Smalltalk/X development environment contains about 5500 classes and some 45000 methods. For a 32bit VM with 4-byte object pointers, roughly 200kB of memory is occupied just by the lookup object reference slots. We think that 200kB (or 400kB for 64bit systems) is acceptable at present.

7.3 Selector Namespaces

As we mentioned in Section 2.1.3, selector namespaces is a valuable mechanism that eases the integration of different pieces of the code. We found this feature particularly useful when a library must be ported from another Smalltalk dialect and runs within Smalltalk/X. As Smalltalk dialects are developed independently, a conflicting method with the same name and different behavior is likely to occur.

In this section, we will describe the implementation of the selector namespaces as implemented in Smalltalk/X 6.1.1β.

7.3.1 Using Selector Namespaces in Smalltalk/X

Smalltalk/X uses an annotation to express that given method belongs to a namespace:

```smalltalk
SequenceableCollection >> atPin: index

"Return the index'th element of me if possible.
Return the first or last element if index is out of bounds."

<namespace: Squeak>

index < 1 ifTrue: [↑ self first].
index > self size ifTrue: [↑ self last].
↑ self at:index
```

A class browser user interface has been adapted to present selector namespace to the programmer visually (Figure 7.2).

The semantics of the selector namespaces in Smalltalk/X differs from Modular Smalltalk namespaces in three ways:

- the method belongs to the namespace or it may belong to no namespace,
- a programming language (except Smalltalk language) of the given method is treated as a namespace if there is no namespace explicitly set and
• a namespace import graph may contain cycles

The No-namespace. To support backward compatibility a method can belong to no namespace. Such methods represent a code as it would behave without any selector namespaces. No-namespace methods are visible from any namespace (if not “overridden” by any “namespaced” method). More precisely: if no method is found among the namespace import graph and class inheritance hierarchy, a no-namespace method is searched. When a message is being sent from no-namespace method, all namespaces are searched. Conceptually, no-namespace can be seen as a special namespace that is automatically imported by all other namespaces and that automatically imports all other namespaces. In a typical instance of Smalltalk/X environment, vast majority of methods do not belong to any namespace.

Programming Language as a Namespace. It may happen – and it often does – that different programming languages define different behavior for the methods with the same name in the same class. For example, in Smalltalk/X an expression $3 \% 2$ results in a complex number $3 + 2i$ whereas in JavaScript or Ruby the same expression results in an integer value of 1 (remainder of 3 divided by 2). Thus programming languages are naturally a sort of namespaces. To ease the coexistence of multiple languages and to maintain interoperability within Smalltalk/X environment programming languages are treated as namespaces which – by definition – import all other programming languages present in the system. This allows programmers to seamlessly call the code written in one language from another one and vice versa. At the same time, it allows programmers to specify different behavior for each language if necessary.

Cycles in Namespace Import Graph. The need for the cycles in the namespace import graph comes from the fact that programming languages as treated as a kind of a namespace that imports all other programming languages. Therefore, there is no reason to restrict regular namespaces from creating import cycles.

In a standard Smalltalk system, i.e., in a system without any selector namespaces, two situations may occur in response to a message send. Either a method is found and executed or the method is not found and doesNotUndertand: message is sent. In a selector namespaces-enabled system the third situation may occur. Consider the following code:

```
1 C >> foo
2    <namespace: NSA>
3     ↑ self bar
4 C >> bar
5    <namespace: NSX>
6     ↑ 10
7 C >> bar
8    <namespace: NSY>
9     ↑ 20
```
Let’s say that the namespace NSA imports both NSX and NSY. Since the namespace imports are not ordered, it is not clear which method to execute in response to a send of the message bar from the method foo. We call this an ambiguous send. In case of an ambiguous send, a receiver is sent #ambiguousMessage:, similarly to #doesNotUnderstand: mechanism.

7.3.2 Implementation

7.3.2.1 Background

First, before discussing the implementation of selector namespaces in Smalltalk/X, we must clarify few terms.

Symbols. Many programming environments have some form of symbols. Lisp calls them “atoms”, in Self the symbols are called “canonical strings”. The symbol is essentially an interned string i.e., for the given string value, there is only one instance symbol at a time. Therefore, two symbols with the same string value are always identical, contrary to the normal strings. Consequently, to check whether two symbols represent the same value, an object identity may be used. The object identity test is much faster than the equality test. In Smalltalk symbols are instances of class Symbol and are often used as method identifiers, class names or symbolic constants.

Selectors. A selector is an object that identifies a method for the VM. The selector along with the receiver and arguments are passed with every message send. In response to a message send, VM searches the receiver’s class for a method with the same selector. Once it is found, the method is executed. If not, the receiver does not implement requested the method and an error is triggered. Note that most Smalltalk VMs including the Smalltalk/X VM compare the selector using an object identity.

Symbols are usually used as selectors. However, Smalltalk VMs usually do not restrict the selectors to be symbols – any object may be used as a selector.

Method name. A method name is an identifier of the method at a programming language level as expressed in the source code. When programmers navigate through an existing code or when they write a new one, they think in terms of the method names.

In Smalltalk, the method names and theirs selectors have the same string value, but generally not true they are not the same. Java, for example, encodes the type names into the selector, therefore the method with name “foo” declared as

```java
public Object foo(int i, String s) ...
```

has a selector with value `foo:(I;Ljava.lang.String;)Ljava.lang.String;`. Moreover, in a single class, for one method name there may more methods with different selectors. For example, for the code

```java
public Object foo(int i, String s) { ... }
```
public Object foo() { ... }

there will be two methods with the same name ("foo") but with different selectors
(foo:(I;Ljava.lang.String;)Ljava.lang.String; and foo:()Ljava.lang.String;).

Method dictionary. In the class-based object oriented languages, methods are usually
stored in a table-like structure within a class metaobject. In Smalltalk such a table is called
a method dictionary. The method dictionary maps a selector to the method metaobject
which holds the executable code. No two methods with identical selectors can be stored
in one method dictionary.

7.3.2.2 Storing Namespaced Methods within a Class

Here presented implementation of the selector namespaces encodes the namespace into the
selector under which the method is installed into the method dictionary. More precisely,
any method in a namespace is installed into the method dictionary under a selector in the
form of "#:<ns>::<original selector>", where ns is a name of the namespace and an
original selector is string value of the selector as it would be used in no-namespace enabled
system. For example, a method "atPin:" in namespace “Squeak” is installed under the
selector #:Squeak::atPin:. Although this solution is far from the ideal one, it does not
require any further changes to the virtual machine. It would probably be better to use
more complex objects instead of pure symbols as keys in the method dictionaries. However,
such change implies relatively complex changes in the existing virtual machine and to the
stc compiler.

7.3.2.3 Selector Namespace-Aware Method Lookup

Figure 7.3 shows a principal implementation of a selector namespace aware lookup method
for Smalltalk/X. The version actually used in the recent version is slightly more complex
in order to optimize common cases and deliver a better performance. First, a selector
namespace the of sending method is determined (lines 8 - 9). Variable namespaces hold
a set of currently searched namespaces. The routine also maintains a set of already visited
namespaces (variable seen) since we allow the cycles in the namespace import graph.

A helper method lookupMethodsForSelector:directedTo:inNamespaces: searches the
class hierarchy starting with the given searchClass for the methods in given namespaces.
If only one method is found, a lookup is finished and this method is returned as a result. If
more than one method is found, each in different namespace, the message send is treated
as ambiguous. If no method is found, imported namespaces are searched in the same way.
If there is no more namespace to search, a “no-namespace” is searched.
7.4 Multi-language Support in Tools

To keep track of programming languages used in the system, a central class `Programming-Language` has been introduced. A language implementer is supposed to subclass this class and fill it with required information like a parser class, compiler class or class used to load the code from a file. A reflective API has been extended to provide programming language information for every piece of code – class or method. Tools then use programming language information to use proper objects to work with the code.

7.5 Summary

To evaluate the concepts presented in the first part of this thesis we improved the Smalltalk/X development platform to support multiple languages better. A metaobject protocol proposed in Chapter 5 has been successfully implemented. We also extended Smalltalk/X environment with the support for selector namespaces that exploits a method lookup MOP. Benchmarks and our experience with everyday usage shows that a VM-level metaobject protocol did not bring about a noticeable performance overhead except for some special cases and – at the same time – it provides powerful and flexible vehicle to extend and modify a given programming language.
Figure 7.2: Smalltalk/X System Browser showing the method in a namespace
| sendingNs sendingMthd queue seen methods |

sendingMthd ← sendingCtx method
sendingNs ← sendingMthd ifNil:[nil] ifNotNil:[sendingMthd nameSpace].

namespaces ← sendingNs == nil ifTrue:[#()] ifFalse:[Array with: sendingNs]. seen ← Set new.

[namespaces notEmpty] whileTrue:
  [] imports |
  imports ← Set new.
  methods ← self
  lookupMethodsForSelector: selector
directedTo: initialSearchClass
inNamespaces: namespaces.
methods size == 1 ifTrue:
  [↑ methods anyOne].
methods size > 1 ifTrue:
  [↑ self ambiguousMessage:
    (Message
      selector: selector
      arguments: argArrayOrNil)].
seen addAll: namespaces.

namespaces do:
  [ namespace |
  namespace ifNotNil:
    [namespace imports do:
      [:import |
        (seen includes: import) ifFalse:
          [imports addAll: namespace import]]].
  namespaces ← imports].

methods ← self
lookupMethodsForSelector: selector
directedTo: initialSearchClass.
methods size == 1 ifTrue:[↑ methods anyOne].

↑ nil

Figure 7.3: Selector namespace aware lookup method
8 SMALLRUBY— a Decent Ruby Implementation

Ruby is a relatively young, general-purpose programming language developed by Yukihiro Matsumoto. The very first version was released in 1994 and it soon became very popular. Ruby is purely object-oriented, dynamically typed scripting language highly influenced by Smalltalk, Perl and Python. Like many other scripting languages, its implementation started with a simple AST-based interpreter often referred as “MRI”. Such an interpreter is easy to implement but its performance tends to be bad when compared to the systems with optimized VMs like JVM or CLR. Nowadays, there are many implementations of Ruby: already mentioned JRuby, Rubinius [27] or IronRuby [3]. As far as we know, almost every alternative implementation of Ruby tries to address performance.

SMALLRUBY is an experimental implementation of the Ruby programming language built on the top of the Smalltalk/X virtual machine. SMALLRUBY served (and it still does) as a testbed for experimenting with multi-language environments. This section gives a brief overview of SMALLRUBY and its implementation.

8.1 Overview

8.1.1 The Ruby Language

Ruby is a very dynamic scripting language, more dynamic than one may think. It has been designed in a human-friendly way, i.e., to be easy to use, to support very agile development, to put almost no restrictions on the code. Yukihiro Matsumoto, Ruby’s author, often says that he is “trying to make Ruby natural, not simple,” in a way that mirrors life. He also adds: “Ruby is simple in appearance, but is very complex inside, just like our human body”. Based on our own experience, we think that everybody who has ever tried to implement an interpreter or compiler for Ruby will agree. Ruby provides few useful syntactic shortcuts to ease programming including:

Default argument values. In many programming languages, the programmer can specify a default argument value as a part of the method definition. If a particular argument is not specified at the callsite, a specified default value is used.

Variadic arguments. In addition to arguments with the default values, Ruby allows a method to be defined with an arbitrary number of arguments. Within the method body, actual arguments are accessible as elements of an argument vector.

Block arguments. In Ruby, every method can be called with or without a “block argument”. Such a block can or can not be specified in the method definition.

Ruby is easy to use but quite complex to implement. Ironically, this makes Ruby to be a nice use case for multi-language environments
Mixins. Ruby supports mixin inheritance. Any class may be mixed with a Ruby module containing the methods. The instances of the resulting class then understand the methods from both the class itself and from the mixed module. Mixins are used to share the code among multiple class hierarchies.

Obviously, the Smalltalk/X VM lacks direct support for such features, because they are missing in the Smalltalk language for which the VM has been designed.

In addition, the coexistence of two languages inside a single environment poses another challenge: a need for different selector behavior for some objects. As Smalltalk and Ruby were developed independently, they define some methods with the same name on the same class, but with different behavior. For instance, in Smalltalk, sending a message / ("division") to an integer with another integer as argument results in either an instance of an integer or fraction, whereas in Ruby the / on the integers always results in an integer value. For interoperability reasons, multiple versions of that method must be able to coexist in a single class. A correct method must be invoked according to the programming language of the sending method.

Although these features may be implemented on the top of the unmodified VM using various tricks, the MOP for the method lookup provides a more elegant solution – easier to implement, understand and change.

8.1.2 SmallRuby

SmallRuby consists of three parts:

- a compiler that compiles the Ruby code directly to a Smalltalk/X bytecode,
- an (incomplete) implementation of the Ruby’s Core Library and
- a thin integration layer that allows programmers to use standard Smalltalk development tools to develop in Ruby.

A great effort has been spent on maintaining the interoperability between Smalltalk and Ruby to allow programmer to mix Smalltalk and Ruby codes almost freely.

8.2 Evaluation and Compilation of Ruby Code

As we said before, Ruby is a highly dynamic language. Dynamic features make Ruby very productive but also difficult to compile in advance. Consider a script in Figure 8.1. The script prints either A#foo or B#foo because the actual superclass of class C is either class

\footnote{Rewriting the selector in the compiler.}
A or class B, depending on the condition on line 13. Therefore the creation of class C must be delayed until the if statement on line 13 is evaluated and done as a part of a script execution.

```ruby
1 class A
2   def foo
3     return "A#foo"
4   end
5 end

7 class B
8   def foo
9     return "B#foo"
10    end
11 end

12 if ((rand(100) % 2) == 0)
13   superclass = A
14 else
15   superclass = B
16 end
17
18 class C < superclass
19 end
20
21 puts C.new.foo()
```

Figure 8.1: Dynamically computed superclass in Ruby

To support this kind of dynamism, SMALLRUBY dissociates evaluation from compilation. When a script is to be executed, it is passed to class Evaluator that reads the code and evaluates it. Occasionally, when a method definition is encountered, it asks a Compiler to byte-compile the method.

### 8.2.1 The Evaluator

The Evaluator is SMALLRUBY’s entry point to evaluate a Ruby code:

```ruby
1 Ruby::Evaluator evaluate:
2   a = 20.factorial()
3   puts "20! = #{a}"
4 
```
The **Evaluator** uses the **Parser** to create an AST from a text representation of the code. Then it evaluates a parsed code, statement by statement. Basically, the **Evaluator** is implemented as an AST visitor following the interpreter design pattern. For example a “visit” method that performs a message send (a method call in Ruby terminology) looks like:

```ruby
1 Ruby::Evaluator >> visitSendNode:sendNode
2 | rec args arrayArgs numPargs selector method |
3 5 rec ← self visit:sendNode receiver.
6 7 | args ← (sendNode args positionalArgs collect:[:arg | self visit:arg ]). |
8 9 numPargs ← args size.
10 11 sendNode args arrayArg ifNotNil:
12 13 | [args ← args , (arrayArgs ← self visit:sendNode args arrayArg)]. |
14 15 numPargs ← numPargs + arrayArgs size.
16 17 sendNode args hasBlockArg ifTrue:
18 19 | [args ← args copyWith: (self visit:sendNode args blockArg)]. |
20 21 args ← args asArray.
22 23 selector ← sendNode selector: numPargs.
24 25 method ← Ruby::Lookup instance
26 27 lookupMethodForSelector:selector directedTo:rec class
28 29 for:rec withArguments:args
30 31 from:context.
32 33 ↑ method
34 35 ifNotNil:
36 37 | [method valueWithReceiver: rec arguments: args] |
38 39 ifNil:
40 41 | (Message selector: selector argument: args)] |
```

Whenever a class definition is reached, the **Evaluator** creates a new class and installs it into the global system dictionary, where all classes are kept:

```ruby
1 Ruby::Evaluator >> visitClassNode: classNode
2 | className class superclass oldReceiver |
3 5 className ← self visit: classNode className
4 6 class ← Smalltalk at: className.
5 7 superclass ← self visit: classNode superClass.
6 8 (superclass isClass or:[superclass isNil]) ifFalse:
7 9 | [ self typeError: |
```
First, a new class name and a superclass are computed (lines 4–6). If a desired class does not exist or a desired superclass is different from the superclass of an already existing class, a new class is created and installed (lines 13–17). Finally, class body is evaluated (line 22). Since in Ruby classes may nest a class stack maintained, the class itself becomes a “self” for the message sends that occur within a class body definition.

Every method definition node is passed to Compiler to create a method:

```ruby
Ruby::Evaluator >> visitDefnNode: defnNode
  | methods |
  methods ← Compiler new
  compileMethod: defnNode
  source: (defnNode sourceFrom: source)
  evaluator: self
  forClass: classStack top.
  install ifTrue:
  [self installMethods: methods to: classStack top].
↑ methods
```

The method #installMethods:to: installs the methods into a particular class.

### 8.2.2 The Compiler

The Compiler is responsible for compiling functions and methods producing a bytecode representation of the method. The Compiler is realized again as a visitor of the method’s
Figure 8.2 shows a “visit method” that compiles a conditional expression.

```ruby
Ruby::Compiler >> visitIfNode:ifNode forEffect:effect
  | endifJumpTarget elsifJumpTarget |
endIfJumpTarget ← self getUniqueJumpTarget:'endIf'.
elsrufJumpTarget ← self getUniqueJumpTarget:'elsIf'.

"Emit code for condition"
self visit:ifNode condition forEffect:CodeEffect forValue.
irBuilder pushLiteral: true; send: #==.
irBuilder jumpAheadTo:elsifJumpTarget if:false.

"Emit true branch"
self visit: ifNode thenBody forEffect:effect.
irBuilder jumpAheadTo:endifJumpTarget.

"Emit false branch"
self visit: ifNode elseBody forEffect:effect.
irBuilder jumpAheadTarget:endifJumpTarget.
```

8.3 Method Lookup in SMALLRUBY

Although a method lookup semantics in Smalltalk and Ruby seem to be similar, there are few differences that make standard Smalltalk method lookup hardly usable for the Ruby code. For instance, Smalltalk/X VM does not support variable a number of arguments (JVM and CLR neither). Ruby’s mixin inheritance requires mixed modules to be searched in addition to the methods defined directly in the class or in one of their superclasses.

SMALLRUBY uses the method lookup MOP to implement Ruby’s method lookup semantics. The SMALLRUBY’s lookup method is shown in Figure 8.3. The lookup is rather complex, but the basic idea of the lookup method is simple. First, it searches the given class for a method that exactly matches the selector, taking the inheritance and mixed modules into account (lines 6 – 9). If no such a method is found, it searches the classes again for a method with the given function name. If a method is found, a new placeholder method is compiled dynamically, installed into a receiver’s class for the future use and then it is returned.
Ruby::Lookup >> lookupMethodForSelector: selector
directedTo: searchClass
from: sendingMethod

| fname method |
method ← NamespaceAwareLookup instance
lookupMethodForSelector:aSelector
directedTo:searchClass
for:aReceiver withArguments:argArrayOrNil
from:sendingContext.
method ifNotNil:[↑ method].

fname ← selector asRubyFunctionName.
method ← self
lookupMethodForName: fname
directedTo: searchClass.
method ifNotNil:
[↑ Ruby::Compiler new
compileProxyForSelector: selector
method: method
inClass: searchClass].
↑ nil.

Figure 8.3: SMALLRUBY’s method lookup implementation
8.4 Language Interoperability

One of the key benefits of a multi-language environment is the possibility to call the code written in one language from the code written in another. Programmers can reuse existing code in that way. A great effort has been spent on Smalltalk and Ruby interoperability and on the integration of Ruby language into a Smalltalk/X platform.

Language interoperability lies in:

- **cross-language method calls** – a possibility to call the code written in language \( A \) from the code in language \( B \) and vice versa.

- **cross-language inheritance** – a possibility to create a language \( A \) class (meaning a class defined by \( A \)) as a subclass of a language \( B \) class and vice versa (assuming that both languages have the notion of classes and inheritance) and,

- **multi-language class composition** – a possibility to define a language \( A \) method (meaning the method written in \( A \)) in a language \( B \) class and vice versa.

Before we explain how SMALLRUBY meets interoperability requirements, we have to explain how Ruby function names are mapped to method selectors (see Section 7.3.2.1 for explanation of the difference between the method name and its selector). As we will explain later, this mapping is a key trick to achieve language interoperability.

SMALLRUBY maps the Ruby method names to the selectors using the following rules:

- If the given method name is one of the Ruby operators, a corresponding selector has the same value as the method name.

- If the given method does not take any arguments, a corresponding selector has the same value as the method name.

- If the given method takes exactly one mandatory or optional argument, a corresponding selector has the same value as the method name concatenated with string “:”.

- If the given method takes \( N > 1 \) mandatory or optional arguments, a corresponding selector has same value as the method name concatenated with the string “:” (the first argument) concatenated \( N - 1 \) times with the string “with:” (e.g. a “with:” is added for every argument except the first one).

- If the given method takes an array argument (in addition to the mandatory and optional ones or block argument), a corresponding selector is constructed as if the method has no array arguments concatenated with the string “withArguments:”.
• If the given method takes block argument (in addition to the mandatory and optional ones or an array argument), a corresponding selector is constructed as if the method has no block argument concatenated with the string “withBlock:”.

Table 8.1 shows examples of mappings. Note that the selectors are used by the VM as the method identification – a selector is passed as an operand of the “send” instruction.

<table>
<thead>
<tr>
<th>Ruby method signature</th>
<th>Corresponding selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>def + (a)</td>
<td>#+</td>
</tr>
<tr>
<td>def &gt;&gt; (a)</td>
<td>#&gt;&gt;</td>
</tr>
<tr>
<td>def foo</td>
<td>#foo</td>
</tr>
<tr>
<td>def foo(a)</td>
<td>#foo:</td>
</tr>
<tr>
<td>def foo(a,b,c)</td>
<td>#foo:with:with:</td>
</tr>
<tr>
<td>def foo(a,b,c = 1)</td>
<td>#foo:with:with:</td>
</tr>
<tr>
<td>def foo(a,b,*c)</td>
<td>#foo:withArguments:</td>
</tr>
<tr>
<td>def foo(a,b,&amp;c)</td>
<td>#foo:withBlock:</td>
</tr>
<tr>
<td>def foo(a,b = 1, *c, &amp;d)</td>
<td>#foo:withArguments:withBlock:</td>
</tr>
</tbody>
</table>

Table 8.1: Examples of mapping Ruby method names to the selectors

Cross-language method calls. Cross-language calls include both calling Ruby from Smalltalk and calling Smalltalk from Ruby. Since in Smalltalk there is no difference between the method name and its selector, a Ruby code can be called directly using Ruby’s method selector. For instance, a Ruby method declared as `def foo(a,b)` on object `a` can be called from Smalltalk like `a foo: arg1 with: arg2`.

The opposite way is a bit more tricky. Smalltalk is one of the few languages where the arguments are interleaved with the method name. Arguments are delimited by a double colon as in `x between: 1 and: 2`. Ruby, on the other hand, follows an Algol tradition: the function name precedes all arguments, separated by a comma. SMALLRUBY syntax has been extended to support double colons to be a part of the method name. A Smalltalk method `#between:and:` on object `a` can be called from Ruby like `a.between:and:(1,2)`. The extended syntax allows double colons to be a part of the method name only in method calls. Double colons cannot be used for the method definition, e.g., SMALLRUBY does not allow Ruby methods to be defined with double colons in the method name.

Cross-language inheritance. In Smalltalk VMs, the data structure describing the class and its structure is a regular object stored in an object heap along with other objects. SMALLRUBY uses the same structures as Smalltalk to store information about Ruby classes. Since the VM knows a little about the objects used as classes (basically it only needs to know how to extract a superclass from the given class object), everything works out of the box.

Multi-language class composition. Here the situation is exactly the same as in the previous case. Everything the VM knows about objects that represent methods is how to
extract the bytecode (which is also an object stored on object heap). Since method objects created by SMALLRUBY are much like Smalltalk methods, Ruby method may be installed into Smalltalk class’s method dictionary and vice versa.

8.5 Current Status

8.5.1 Standards Compliance

Whenever a new implementation of a Ruby language comes up, the very first question is: “Does it run Rails?” [64]. In case of SMALLRUBY the short answer is: “No, it does not.”

The long answer is: Nowadays, SMALLRUBY is far from the complete implementation of Ruby language. One of the reason is the fact that Ruby comes up with an extensive class library which has not been reimplemented nor ported yet. The implementation of the whole class library requires a lot of resources by means of time and effort need to accomplish that task.

The primary goal of a SMALLRUBY development was to create a language to experiment with and a tool to prove our ideas on multi-language environments.

8.5.2 Performance Benchmarks

Ruby’s reference implementation – MRI – lacks of the performance. One of the main ideas behind any alternative implementation of Ruby is to use better virtual machine to provide better performance.

Figure 8.4 shows the results of selected standard Ruby benchmarks running on MRI (original, reference Ruby implementation), YARV [43], JRuby 1.5 and SMALLRUBY. For full result data refer to appendix A.

Discussion. Except for the three cases SMALLRUBY provides in the order of magnitude better performance than MRI. This is not surprising since MRI is implemented as simple, AST-based interpreter and therefore it tends to be slow.

In most cases SMALLRUBY is 2-5 times faster than YARV. YARV VM is implemented as a pure bytecode interpreter whereas the Smalltalk/X VM translates a bytecode into a native code which is, in turn, executed on the bare hardware.

SMALLRUBY also outperforms JRuby in most benchmarks. Although JVM employs most advanced optimization techniques like a customized compilation or method inlining, the need to simulate the Ruby method lookup at the object level significantly degrades the overall performance. SMALLRUBY takes an advantage of a VM-level method lookup MOP. Therefore, the Ruby method dispatch is as fast as the Smalltalk method dispatch. Thanks

[^3]: Located in benchmarks/ subdirectory of MRI source code
Figure 8.4: Selected Ruby Performance Benchmarks
to using ILCs, a method dispatch in SMALLRUBY is reduced to just a few machine instructions (in case the method is cached).

There are four cases, where SMALLRUBY performance is poor:

- **bm_app_raise.rb** and **bm_app_rescue.rb**. These benchmarks test the exception handling. Exceptions are much slower in SMALLRUBY because they employ a Smalltalk exception handling mechanism. In Smalltalk, exceptions are traditionally implemented purely on the object level (i.e., with no or little support in the VM). Moreover, the semantics of the exceptions in Smalltalk is more complex than in Ruby or Java, so the overhead connected with the exception handling in Smalltalk is bigger.

- **bm_vm2_array.rb**. This benchmark tests the creation of literal arrays. The reason for poor performance of this particular benchmark is that the SMALLRUBY’s compiler does not optimize creation of purely literal arrays, i.e., such a literal array whose elements are all literals, whereas YARV and JRuby does.

- **bm_vm2_poly_method.rb**. This benchmark measures the performance of polymorphic send. It is not clear what causes such a poor performance when compared to JRuby. The detailed analysis showed that the Ruby lookup is called only for the first two method calls. In all subsequent calls the lookup routine has not been called since a PIC cached the result. Therefore the poor performance is certainly not caused by the method lookup MOP. Further investigation of this bad behavior requires an extensive profiling on the hardware level.

### 8.6 Summary

In this chapter, we briefly described SMALLRUBY— an alternative implementation of the Ruby programming language built on the top of Smalltalk/X virtual machine. SMALLRUBY consists of, first, a compiler that compiles Ruby code into the bytecode understood by the Smalltalk/X VM and second, a thin layer that integrates a Ruby language into a Smalltalk/X development environment. SMALLRUBY also provides a great degree of interoperability. No glue code is required to call the Ruby code from Smalltalk or vice versa. A Ruby method can be defined in a Smalltalk class and vice versa. Benchmark results show that SMALLRUBY outperforms other implementations of the Ruby language. In particular, it is faster than JRuby – a Ruby implementation for the JVM. This is because SMALLRUBY uses MOP proposed in Chapter 5. MOP enables a method dispatch to execute at full speed, eliminating the need for slower object level simulation.

Although SMALLRUBY is far from complete Ruby the implementation, its performance and ability to cooperate with Smalltalk code is very promising. Future work will focus on the improvements of the compiler, Ruby’s standard library and tools.
9 Conclusion

In this chapter we summarize our work done during the past few years and we provide the summary of achievements accomplished during this period. We also outline directions for the future research.

9.1 Summary

In the first two chapters of this thesis we argued for the need for multi-language programming environments. We have also shown that, despite all the effort, a current virtual machine support for hosting multiple languages is limited. Indeed, the coexistence and cooperation of multiple languages within a single environment involves wide variety of problems.

In this thesis we have shown that

- it is possible to integrate several programming languages that use several different execution runtimes (a virtual machine and AST interpreter, in particular) into a single, consistent programming environment, while preserving the user experience in a sense of the development tools and,

- it is possible to push a part of the language semantics traditionally implemented within the virtual machine (the method lookup algorithm in particular) out of the virtual machine while preserving the same execution speed.

Benchmarks and practical experience with the real world applications that use the techniques we developed support our claims.

9.2 Contributions

Contributions of this thesis are the following ones:

Chapter 3. This chapter presents a debuggable interpreter design pattern – an extension of well known interpreter design pattern. It extends the interpreter with a small set of hooks allowing a debugger service to intercept a program interpretation. The debugger service models debugging operations by means of debugger mode. Easy to implement and open to extensions are the main benefits of a debuggable interpreter design pattern.

Chapter 4. Debuggers are tools used to test and debug other programs. Although modern programming environments provide sophisticated symbolic debuggers, they fail when debugging application written in multiple languages. More precisely, they cannot handle well transitions between languages. Chapter 4 describes the architecture of an unified
debugger – a debugger that is able to debug programs written in multiple programming languages using different execution runtimes.

**Chapter 5.** This chapter addresses inflexibility of virtual machines to handle different semantics of message sends efficiently. A *metaobject protocol for the defining custom method lookup algorithms* has been designed and implemented. Main benefits of the proposed MOP consist of: first, simple to use, second, ability to define a custom method lookup algorithm for every single class and method, third, backward compatibility and finally the design according to optimizations employed to speed up a message dispatch.

**Chapter 6.** A brief description of PERSEUS– a framework for the language construction that employs debuggable interpreter pattern – is discussed in this chapter. Various extensions to the debuggable interpreter design pattern are described, i.e., conditional breakpoints, an execution tracer or a simple profiler are shown.

**Chapter 7.** A Smalltalk dialect – Smalltalk/X has been adapted to handle multiple languages in order to test and evaluate achievements described above. Besides other changes we made to Smalltalk/X, the recent version contains a fully functional implementation of the MOP described in chapter 5. This chapter presents our experience with such environment together with the performance analysis of MOP-enabled VM.

**Chapter 8.** Last chapter describes SMALLRUBY– our experimental implementation of a Ruby programming language for Smalltalk/X virtual machine developed during our research. Smalltalk/X running SMALLRUBY is a real-world example of an environment with two programming languages running seamlessly within one virtual machine. Performance benchmarks prove that the method lookup MOP enhances the ability of the VM to host multiple languages efficiently.

### 9.3 Directions for Future Research

**Language Symbiosis and Interoperability.** In previous chapter we described mechanisms that ease the implementation of multiple programming languages on the top of one virtual machine. Although we developed solutions to technical, implementation-related problems, but still there are many open issues at the program semantics level. If a language A calls method in language B, what language method dispatch semantics shall be used? Languages must be isolated from each other to some extent (because of conflicting behavior). On the other hand, we need to integrate them in order to be able to use them in a single application. The question on the way modelling a proper isolation and providing interoperability open a new area for further research.

**User-level Combinators for Lookup Methods.** Chapter 5 describes only a VM part of the MOP. There is no or little support at the object level. One particularly interesting point is how to combine multiple lookup routines that are not aware of each other. In other words: how to cope with several unanticipated modifications of a method lookup routine? For example, for one particular language one can need to load trait support or...
selector namespaces support or both if desired, depending on the application needs. Hence combinator

dors for the method lookup would make whole environment even more flexible. This is another interest-
ing direction for future research.

**Reified, generalized ILC/PIC.** As we have shown in Chapters 5 and 7, inline caches and polymorphic inline caches are a key runtime optimization technique for achieving fair per-
formance. Hence ILC and PIC reification seems to be a useful and important step forward. The access to ILC and PIC will ease a dynamic analysis of the applications. Custom just-in-time compilers need an access to the PIC data in order to generate opti-
mized machine code (to inline methods or to emit custom code for different receivers, for instance).

ILCs and PICs optimizes a method dispatch based on the class of the first argument (the receiver). It should be also possible to generalize them to be able to optimize the dispatch on any argument of the message (not only the first one) and using any property (not only arguments’ class). Generalized ILCs and PICs exposed to the user through the code reflection facilities together with the MOP support would be a very interesting feature. For example, programmers can use MOP (similar to the one described in Chapter 5) to implement a multiple dispatch logic or even a predicate dispatch logic in a very efficient way. In the future, we are going to explore this area, as well.
10 Bibliography

   http://eclipse.org/aspectj/.

   http://sourceware.org/gdb/.

   http://ironruby.net/.


   Revue des Sciences et Technologies de l’Information (RSTI) – L’Objet (Numéro spécial :  


    in Java. In Proceedings of 20th International Conference on Object-Oriented Pro-  
    gramming, Systems, Languages, and Applications (OOPSLA’05), pages 177–189, New  

[12] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: Controlling visibility  
    of class extensions. Technical Report IAM-04-003, Institut für Informatik, Universität  
    Bern, Switzerland, June 2004.

    of class extensions. Computer Languages, Systems and Structures, 31(3-4):107–126,  


[58] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, Nov. 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.


http://www.squeak.org/.


http://www.is-research.de/info/vmlanguages/.


11 Refereed publications of the author


12 Unrefereed publications of the author


13 Other work of the author


Part III

Appendices
A Benchmark Results

A.1 Notes

Within our benchmarks we measure real time taken to run a particular benchmark. To eliminate external effects (such as interleaving execution of the benchmark by another operating system process) each benchmark has been run several times and the resulting mean value is computed only from a relevant sample.

Relevant sample $T_g$ is computed as follows:

1. Let $T_a = \{t_1, t_2, \ldots, t_n\}$ be a sample consisting of all observations. Then

$$\text{Mean}_a = \frac{1}{n} \sum_{t_i \in T_a} t_i$$

is the arithmetic mean of sample $T_a$ and

$$\sigma_a = \sqrt{\frac{1}{n} \sum_{t_i \in T_a} (\text{Mean}_a - t_i)^2}$$

is standard deviation of sample $T_a$.

2. Relevant sample $T_g = \{t_i \in T_a\}$ such that $|t_i - \text{Mean}_a| \leq 3\sigma_a$. 
## A.2 MOP Benchmarks

### A.2.1 Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Configuration</th>
<th>Time [ms]</th>
<th>Mean [ms]</th>
<th>Slowdown [(^\d)]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unimorphic send</strong></td>
<td>STD VM</td>
<td>320</td>
<td>308</td>
<td>357</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>312</td>
<td>317</td>
<td>326</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>304</td>
<td>308</td>
<td>329</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>306</td>
<td>307</td>
<td>326</td>
</tr>
<tr>
<td><strong>Polymorphic send 2</strong></td>
<td>STD VM</td>
<td>3526</td>
<td>4320</td>
<td>3908</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>3500</td>
<td>4355</td>
<td>3905</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>3474</td>
<td>4394</td>
<td>3905</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>3468</td>
<td>4371</td>
<td>3905</td>
</tr>
<tr>
<td><strong>Polymorphic send 4</strong></td>
<td>STD VM</td>
<td>4100</td>
<td>4718</td>
<td>4683</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>4085</td>
<td>4718</td>
<td>4683</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>3972</td>
<td>4634</td>
<td>4506</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>3968</td>
<td>4634</td>
<td>4506</td>
</tr>
<tr>
<td><strong>Polymorphic send 20</strong></td>
<td>STD VM</td>
<td>5234</td>
<td>5250</td>
<td>5295</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>5234</td>
<td>5250</td>
<td>5295</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>5111</td>
<td>4980</td>
<td>4824</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>5107</td>
<td>4974</td>
<td>4824</td>
</tr>
<tr>
<td><strong>Polymorphic send 512</strong></td>
<td>STD VM</td>
<td>5998</td>
<td>6103</td>
<td>6099</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>5998</td>
<td>6103</td>
<td>6099</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>5611</td>
<td>5379</td>
<td>5320</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>5419</td>
<td>5379</td>
<td>5320</td>
</tr>
<tr>
<td><strong>Web Application</strong></td>
<td>STD VM</td>
<td>2206</td>
<td>2798</td>
<td>2333</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>2251</td>
<td>2055</td>
<td>2223</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>2151</td>
<td>2433</td>
<td>2404</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>2147</td>
<td>2404</td>
<td>2404</td>
</tr>
<tr>
<td><strong>File Browser</strong></td>
<td>STD VM</td>
<td>787</td>
<td>545</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>1386</td>
<td>1005</td>
<td>1094</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>1051</td>
<td>1027</td>
<td>1106</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>895</td>
<td>968</td>
<td>1068</td>
</tr>
<tr>
<td><strong>Class Browser</strong></td>
<td>STD VM</td>
<td>13679</td>
<td>11128</td>
<td>11128</td>
</tr>
<tr>
<td></td>
<td>MOP VM no lookup obj</td>
<td>37925</td>
<td>37144</td>
<td>36991</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj C optimized</td>
<td>64246</td>
<td>63529</td>
<td>64185</td>
</tr>
<tr>
<td></td>
<td>MOP VM std lookup obj pure smalltalk</td>
<td>10156</td>
<td>98369</td>
<td>10156</td>
</tr>
</tbody>
</table>

### Benchmark
- for the description of benchmarks see Section 7.2.2.1

### Configuration
- for the description of configurations see Section 7.2.2.1

### Time
- time required to run a given benchmark in milliseconds

### Mean
- an arithmetic mean value from relevant sample

### Slowdown
- how many times is the given configuration slower than reference (STD VM)

---

Table A.1: MOP Benchmark results
A.2.2 Benchmark code

`From Smalltalk/X, Version:6.1.2 on 29-08-2010 at 09:53:27 PM` !

`"{ NameSpace: Benchmarks }"`

Object subclass: #MOP
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'benchmarks-Misc'
!
MOP class instanceVariableNames:
'initialized
objectArray1 objectArray2 objectArray4
objectArray20 objectArray63 objectArray512'

No other class instance variables are inherited by this class.

```
MOP class methodsFor: 'initialization'!
init
  initialized = true ifTrue:[^self].
  self initBenchmarks.
  self initObjectArrays.
  ObjectMemory scavenge.
  ObjectMemory tenure.
  initialized := true.
!`

initBenchmarks
Smalltalk loadPackage: 'stx:goodies/aida'.
Aida::SwazooAida startOn: 8888.
!

initObjectArray: size classes: nClasses
| classes array |
classes := (1 to: nClasses) collect:
  [:idx | | cls |
  cls := ClassDescription new.
  cls compile: ('foo ^ %1' bindWith: idx).
  cls].
array := Array new: size.
1 to: size do:
  [:idx |
  array at: idx put: (classes at: (idx \ nClasses) + 1) new].
'array'
!

initObjectArrays
| size |
sizes := 1024.
objectArray1 ifNil: [
  objectArray1 := self initObjectArray: sizes first.
  objectArray2 := self initObjectArray: sizes second.
  objectArray4 := self initObjectArray: sizes third.
  objectArray20 := self initObjectArray: sizes fourth.
  objectArray63 := self initObjectArray: sizes fifth.
  objectArray512 := self initObjectArray: sizes sixth.
]
!
MOP class methodsFor: 'benchmarks - macro'!

fileBrowser
| b |
b := FileBrowserV2 new.
b open.
b currentDirectory: '/home/jv/Projects/SmalltalkX' asFilename.
b currentFilenameHolder value: ('/home/jv/Projects/SmalltalkX' ,
  'cvs-sync-log-2009-08-19.txt' ,
  asFilename) asCollection.
b newTextEditor.
b close.
newSystemBrowser
| b |
b := Tools::NewSystemBrowser new.
b open.
b switchToClass: Array selector: #storeArrayElementOn.
b switchToClass: Tools::NewSystemBrowser selector: #classListGenerator.
b switchToClass: Tools::NewSystemBrowser selector: #spawnSenderChain:.
b spawnSenderChainBuffer.
b close.

webApp
10 timesRepeat:
[(HTTPInterface get:'/' fromHost:'localhost' port:8888) data size]

MOP class methodsFor:'benchmarks - micro'!
polymorphic_2
1024 timesRepeat:[objectArray2 do[:o|o foo]]

polymorphic_20
1024 timesRepeat:[objectArray20 do[:o|o foo]]

polymorphic_4
1024 timesRepeat:[objectArray4 do[:o|o foo]]

polymorphic_512
1024 timesRepeat:[objectArray512 do[:o|o foo]]

polymorphic_63
1024 timesRepeat:[objectArray63 do[:o|o foo]]

unimorphic
1024 timesRepeat:[objectArray1 do[:o|o foo]]

MOP class methodsFor:'lookup'!
lookupMethodForSelector:selector directedTo:initialSearchClass
for:aReceiver withArguments:argArrayOrNil
from:sendingContext
| cls md method |
cls := initialSearchClass.
[ cls notNil ] whileTrue:
[ md := cls methodDictionary.
method := md at:selector ifAbsent:nil.
method notNil ifTrue:[ ^ method ].
cls := cls superclass].
^ nil

MOP class methodsFor:'running'!
run: label

* 
Object setLookupObject: nil.
sel run: 'MOP VM, no lookup obj'.
Object setLookupObject: Lookup new.
sel run: 'MOP VM, std lookup obj - C optimized'.
Object setLookupObject: self.
sel run: 'MOP VM, std lookup obj - pure smalltalk'.
sel run: 'STD VM'
<table>
<thead>
<tr>
<th>s</th>
</tr>
</thead>
</table>
| s := '/home/jv/Research/PhD thesis/benchmarks/MOP.csv'
| '/tmp/MOP.csv'
| asFilename appendStream.
| self init.
| [ |
| | uni: | mon: | poly: | poly: | webApp |
| | morphic_2 | morphic_4 | 20 | 512 |
| fileBrowser newSystemBrowser
| do: |
| [:bench] | self put: label to: s.
| | self put: bench to: s.
| | 6 timesRepeat:[self put: (self time: bench) to: s].
| | s cr].
| ] ensure: |
| | s close
| |
| !
| MOP class methodsFor:'utils'!
| put: o to: s
| s nextPut: o.
| o printOn: s.
| s nextPut: $.
| s space.
| s flush
| |
| !
| time: bench
| ObjectMemory flushCaches.
| "Time millisecondsToRun:[20 timesRepeat:[self perform: bench]].
### APPENDIX A. BENCHMARK RESULTS

#### A.3 Ruby Benchmarks

##### A.3.1 Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Implementation</th>
<th>Mean [ms]</th>
<th>Speedup [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bm_app_base.rb</code></td>
<td></td>
<td>2279</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure.rb</code></td>
<td></td>
<td>4898</td>
<td>2.68</td>
</tr>
<tr>
<td><code>bm_app_new.rb</code></td>
<td></td>
<td>3062</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_optimize.rb</code></td>
<td></td>
<td>2156</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure2.rb</code></td>
<td></td>
<td>2062</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure3.rb</code></td>
<td></td>
<td>2079</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure4.rb</code></td>
<td></td>
<td>2062</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure5.rb</code></td>
<td></td>
<td>2062</td>
<td></td>
</tr>
<tr>
<td><code>bm_app_ensure6.rb</code></td>
<td></td>
<td>2062</td>
<td></td>
</tr>
</tbody>
</table>

#### Legend:

- **T1** – arithmetic mean from relevant sample
- **Mean** – how many times is the given implementation faster than reference (Ruby 1.8 – MRI)

Table A.2: Ruby benchmark results (part 1)
## Appendix A. Benchmark Results

### Table A.3: Ruby benchmark results (part 2)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Implementation</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>Mean</th>
<th>Speedup [1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bm_vm2_send.rb</td>
<td>Ruby 1.8</td>
<td>5714.308</td>
<td>5695.388</td>
<td>5759.699</td>
<td>5712.201</td>
<td>5782.279</td>
<td>5830.83</td>
<td>5749.11</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>1364</td>
<td>1341</td>
<td>1338</td>
<td>1441</td>
<td>1403</td>
<td>1349</td>
<td>1359.17</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>659188479</td>
<td>669128232</td>
<td>654367486</td>
<td>693319269</td>
<td>642551609</td>
<td>644274696</td>
<td>6559.09</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>365</td>
<td>848</td>
<td>685</td>
<td>811</td>
<td>744</td>
<td>696</td>
<td>785</td>
<td>1419</td>
</tr>
<tr>
<td>bm_vm2_swap.rb</td>
<td>Ruby 1.8</td>
<td>17854.462</td>
<td>18454.318</td>
<td>18589.984</td>
<td>19899.017</td>
<td>19997.38</td>
<td>19694.343</td>
<td>18793.92</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>8187</td>
<td>8222</td>
<td>8295</td>
<td>8498</td>
<td>8262</td>
<td>8252</td>
<td>6241.84</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>1546123771</td>
<td>1432109982</td>
<td>1436351318</td>
<td>1341833889</td>
<td>1346999631</td>
<td>134175697</td>
<td>1324.99</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>627</td>
<td>627</td>
<td>627</td>
<td>627</td>
<td>627</td>
<td>627</td>
<td>627</td>
<td>26.87</td>
</tr>
<tr>
<td>bm_vm2_swap.rb</td>
<td>Ruby 1.8</td>
<td>7007.1</td>
<td>7677.434</td>
<td>7597.469</td>
<td>7709.961</td>
<td>8274.712</td>
<td>8142.135</td>
<td>7834.8</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>1578</td>
<td>1596</td>
<td>1589</td>
<td>1555</td>
<td>1512</td>
<td>1456</td>
<td>1386.99</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>1591239122</td>
<td>1551491732</td>
<td>1566722222</td>
<td>1614213229</td>
<td>1597609534</td>
<td>1615785586</td>
<td>1527.73</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>198</td>
<td>198</td>
<td>198</td>
<td>198</td>
<td>198</td>
<td>198</td>
<td>198</td>
<td>20.63</td>
</tr>
<tr>
<td>bm_memcpy.rb</td>
<td>Ruby 1.8</td>
<td>13103.999</td>
<td>13078.684</td>
<td>13154.761</td>
<td>13072.562</td>
<td>12911.196</td>
<td>13140.464</td>
<td>13059.39</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>2492</td>
<td>2492</td>
<td>2492</td>
<td>2492</td>
<td>2492</td>
<td>2492</td>
<td>2492</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>2126684852</td>
<td>214768687</td>
<td>2124072352</td>
<td>2134170562</td>
<td>2148858286</td>
<td>2132325001</td>
<td>2187.27</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
<td>1152</td>
</tr>
<tr>
<td>bm_memcpy.return.rb</td>
<td>Ruby 1.8</td>
<td>35620.274</td>
<td>34728.285</td>
<td>35498.956</td>
<td>34969.256</td>
<td>35196.784</td>
<td>35436.531</td>
<td>35576.14</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>4535</td>
<td>4535</td>
<td>4535</td>
<td>4535</td>
<td>4535</td>
<td>4535</td>
<td>4535</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>2641503342</td>
<td>2726803862</td>
<td>2847549779</td>
<td>2872683894</td>
<td>2619049894</td>
<td>295085876</td>
<td>28628.92</td>
</tr>
<tr>
<td>bm_memcpy_block.rb</td>
<td>Ruby 1.8</td>
<td>35114.41</td>
<td>35613.014</td>
<td>35653.815</td>
<td>35742.526</td>
<td>35807.601</td>
<td>35117.181</td>
<td>35584.21</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>10982</td>
<td>11003</td>
<td>11120</td>
<td>11184</td>
<td>11145</td>
<td>11113</td>
<td>11134.85</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>3411306438</td>
<td>3427525678</td>
<td>3423482917</td>
<td>3541979794</td>
<td>3081476664</td>
<td>3417590803</td>
<td>3490.72</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>1058</td>
<td>985</td>
<td>939</td>
<td>916</td>
<td>917</td>
<td>893</td>
<td>956.33</td>
<td>36.98</td>
</tr>
<tr>
<td>bm_memcpy_block.rb</td>
<td>Ruby 1.8</td>
<td>16611.667</td>
<td>16541.833</td>
<td>16469.624</td>
<td>16576.815</td>
<td>16623.570</td>
<td>16633.255</td>
<td>16633.56</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>8941</td>
<td>8941</td>
<td>8941</td>
<td>8941</td>
<td>8941</td>
<td>8941</td>
<td>8941</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>2722473421</td>
<td>2614733292</td>
<td>2853596824</td>
<td>2876835436</td>
<td>2621039432</td>
<td>2625497041</td>
<td>2285.54</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>894</td>
<td>894</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
</tr>
<tr>
<td>bm_memcpy_alloc.rb</td>
<td>Ruby 1.8</td>
<td>9991839</td>
<td>10000.451</td>
<td>9978.652</td>
<td>9971.861</td>
<td>10100.147</td>
<td>9966.848</td>
<td>10000.16</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>3148</td>
<td>3148</td>
<td>3148</td>
<td>3148</td>
<td>3148</td>
<td>3148</td>
<td>3148</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>961362959</td>
<td>960384768</td>
<td>988238451</td>
<td>960912299</td>
<td>967101418</td>
<td>96633927</td>
<td>9661.58</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>894</td>
<td>894</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
<td>889</td>
</tr>
<tr>
<td>bm_memcpy_method.rb</td>
<td>Ruby 1.8</td>
<td>24280.579</td>
<td>23971.479</td>
<td>23850.335</td>
<td>23772.269</td>
<td>23118.084</td>
<td>23748.541</td>
<td>23967.11</td>
</tr>
<tr>
<td></td>
<td>Ruby 1.9</td>
<td>2381</td>
<td>2445</td>
<td>2438</td>
<td>2424</td>
<td>2424</td>
<td>2424</td>
<td>2424</td>
</tr>
<tr>
<td></td>
<td>JPG 1.7</td>
<td>3829764255</td>
<td>3710079435</td>
<td>3898411336</td>
<td>3074214485</td>
<td>3417084352</td>
<td>4432.32</td>
<td>5.54</td>
</tr>
<tr>
<td>SmallRuby</td>
<td>845</td>
<td>845</td>
<td>845</td>
<td>845</td>
<td>845</td>
<td>845</td>
<td>845</td>
<td>845</td>
</tr>
</tbody>
</table>

### Legend:

- **T1...T6** - time required to run given benchmark in milliseconds
- **Mean** - arithmetic mean from relevant sample
- **Speedup** - how many times is the given implementation faster than reference (Ruby 1.8 – MRI)
B Lists of abbreviations

VM Virtual Machine
JVM Java Virtual Machine
CLR Common Language Runtime
API Application Programming Interface
ABI Application Binary Interface
MOP Metaobject Protocol
ILC Inline Cache
PIC Polymorphic Inline Cache
AST Abstract Syntax Tree
UI User Interface
DSL Domain-specific Language