Persephone

Περσεφόνη

Taking Smalltalk Reflection to the sub-method Level

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Philippe Michael Marschall

2006

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Further information about this work, the tools used and an online version of this document can be found at the following places.

Philippe Marschall
Stritenstrasse 40
CH-3176 Neuenegg
kustos@gmx.net
http://www.squeaksource.com/JCompiledMethods.html

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://www.iam.unibe.ch/~scg/

Abstract

Smalltalk traditionally has good support for structural reflection. This comes from the fact that classes are first class, high level objects. This reflection support has allowed Smalltalk implementations to build tools decades ago that surpass those of many other languages today. These tools are basically a user interface for introspection and intercession. The reflective facilities of Smalltalk are not only used by tools but also by Smalltalk developers for metaprogramming.

However the Smalltalk reflection support stops at the method border. The only first class models for reflection at the sub-method level Smalltalk supports are collections of bytes or characters. This prevents tools from truly looking into the method and makes it hard to create a new generation of tools that go beyond the five pane browser and work at the sub-method level. It also prevents Smalltalk developers from doing metaprogramming at a sub-method level.

We present reflective methods: a first class, high level abstraction of a method that supports rich structural reflection at the sub-method level and show how it eases metaprogramming and the creation of tools at the sub-method level such as a pluggable type checker.

Acknowledgements

First I wish to thank my supervisor Marcus Denker for his guidance, technical expertise and overall help.

Also I would like to thank Prof. Dr. Oscar Nierstrasz, head of the Software Composition Group, for giving me the possibility to do my master's thesis in his group.

Then I would like to thank all the other people who contributed to the success of this thesis including all members of the SCG group. This especially goes for Nik Haldimann, Adrian Lienhard and Stefan Reichhart who used the work I did, helped me to improve it and gave me case studies. I would also like to thank Marcus Denker, Prof. Dr. Stéphane Ducasse, Orla Greevy, Adrian Lienhard and Prof. Dr. Oscar Nierstrasz for reading early versions of this thesis and their comments that helped me improve it.

Many thanks to my family for all their support and encouragements during my studies.

Last but not least I would like to thank Klaus D. Witzel for unintentionally giving me the introduction story.

Philippe Marschall
December 2006

Contents

Chapter 1

Introduction

Recently someone asked on the squeak-dev[1] mailing list how to find all senders of a `#to:do:` which are inlined by the compiler[2]. He was told to "Just have a look what the decompiler does.".

How did it happen that this is the preferred way of doing such things in Smalltalk? Smalltalk prides itself by pioneering code browsers and refactoring support. You are regularly told that in Smalltalk "everything is an object", there are no source files and that programming is in fact only using reflection.

Why can none of these tools or objects can help him for this task? The tools Smalltalk provides use the abstractions for code present in the image. In the case of classes this works well. Classes provide a high level abstraction and convenient methods. Introspection and intercession for classes works well and is easy to use. This is why Smalltalk code browsers are so good at dealing with classes and why metaprogramming with classes in general is so easy in Smalltalk. The situation is different for methods. The tool or metaprogrammer can choose between two low level abstractions: text and bytecode. Both have poor support for introspection and intercession.

The compiler uses its own model for methods. But this model is used by no other tool. To decide whether or not to inline a `#to:do:` the compiler has its own rules that are visible to no other tool. That tools do not share a common, high level model makes writing custom tools hard because communication has to happen at a very low level of abstraction. It also makes metaprogramming at the sub-method level hard because only very little information is available at this low level.

Methods in Smalltalk have many properties of source files in languages that

[1] `http://lists.squeakfoundation.org/cgi-bin/mailman/listinfo/squeak-dev`

[2] Inlining is an optimization where the compiler eliminates certain message sends.

do not have the concept of an image. They act as a compilation unit, a string is fed into the compiler and a byte array is returned. This results in a situation comparable to a hypothetical file based language where a source file contains just one method.

Compared to methods classes in Smalltalk support reflection and are causally connected to the system. If we send a message to a class like `#addSelector:withMethod:` then this changes the system. This has led to the adoption of classes as the first class model for tools. We use the term *reflective method* for methods offer a high level of abstraction and support reflection similar to classes. The goal of this thesis is to implement reflective methods and show how they ease metaprogramming at the sub-method level and the creation of tools.

1.1 Contributions

The contributions of this thesis are:

- To identify some of the problems that arise for tools and metaprogramming when code is treated as text or a byte array.
- To identify requirements for reflective methods.
- To provide an implementation of reflective methods.
- To show examples how reflective methods improve the current situation, simplify metaprogramming and tool writing and even make a new generation of advanced tools possible.

1.2 Outline

- Chapter 2 outlines the general problem of the code representation in Squeak [IKM+97] and in a Smalltalk system in general.
- Chapter 3 shows other work that either uses a higher level of abstraction for code or would profit from using one.
- Chapter 4 describes requirements for reflective methods.
- Chapter 5 presents our constrained implementation of reflective methods.
- Chapter 6 validates our claims about the benefits of reflective methods by implementing several tools using reflective methods. Some of these

were built only within the context of this project while others are used in real world projects.

- Chapter 7 identifies future work.
- Chapter 8 concludes by outlining our experiences while implementing reflective methods.

Chapter 2

Problem

For classes Smalltalk provides objects that offer a high level of abstraction and good reflection support. This eases the creation of tools as well as metaprogramming. However the high level of abstraction and good support for reflection stops at the method boundary. The problems at the sub-method level can be summarized as:

- The two first class representations for methods are low level abstractions and provide only very limited reflection support.
- High level models for methods exist but they are not causally connected.
- No common, high level, extensible model for methods exists that is supported by all tools.
- Communication and collaboration between tools has to use one of the two low-level representations.
- During the transformation to a low level representation, all not directly supported information is lost.

In the following we will first introduce the abstractions and reflection support Smalltalk provides for classes. Then we will present the abstractions for code that exist at the method level and below, what reflective facilities they offer and what problems they have. In the end we will show the consequences for building tools and metaprogramming.

First Class Abstractions

One abstraction for code at the sub-method level is a `String`. This is the code the programmer typed. This abstraction is mainly used by source code

management tools and by the browsers to display code for the programmer. Support for introspection and intercession is limited at best because it is just a `Collection` of `Character`s. If any higher level abstraction is needed it has to be built. Even tasks like converting the assignment operators from "_" to ":=" turned to be out a real challenge in Squeak because a "_" string does not have any contextual information. It could be an assignment operator, but it could just as easily be a part of a string literal or a comment. Substring replacements will not work and custom scanning code had to be written.

The second abstraction is the `CompiledMethod`. It is built for the bytecode interpreter of the VM. It is an array of bytecodes and its interface is made for dealing with bytecodes. Thus it has many methods like `#initialPC` and `#endPC` that are needed for the bookkeeping of the VM. Introspection generally includes either bit twiddling or parsing bytes with `InstructionStream`. Intercession support is practically non-existent.

High Level Abstractions

An *abstract syntax tree* (AST)[1] is the most common high level abstraction for code. An AST is a tree where each node represents a syntactical element like a message send or an assignment. Only variables and literals can be leaf nodes.

The ability to create an AST from source code alone does not solve the problem. ASTs are not the first class representation, they are only an ad hoc view, as they are not causally connected to the system. This means that changes to it do not take effect, they have to be transformed into a low level representation to achieve a change in the system. An AST is also not the common model of a method for tools which means it can not be used to communicate or share metainformation between tools. This results in the following situation when performing a refactoring: first the source code string is parsed by the parser of the refactoring engine and an AST is created. This AST is used to perform the refactoring. Then the AST is transformed back into a source code string. This new source code string is parsed by the parser of the compiler and a different kind of AST is created which is finally transformed into bytecode. ASTs can however become part of the solution once they are causally connected and extensible enough to be adopted as first class model of methods for tools.

There are several different AST implementations available in Squeak.

[1] http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html

- An advanced one is the Refactoring Browser AST [RBJ97] (RB AST). The NewCompiler [Han] and all the tools working on the RB stack (RB, SLint, search tool, rewrite tool) use it.
- The CodeModel[2] AST models whitespace but captures just the structure of the code and includes only limited behavior for querying and transforming.
- Finally there is a less powerful AST used by the old compiler.

Other Abstractions

Intermediate Representation (IR) is a thin abstraction layer over bytecode that represents bytecodes structure as a tree. The NewCompiler uses it as an intermediate layer between the AST and bytecode. Additionally, a representation like IR can be used by code transformation tools like ByteSurgeon [DDT06] to modify binary code. Although this can be used for reflection at the sub-method level it has several disadvantages. First, it is difficult to distinguish between instructions that where added by the code transformation tool and original instructions. Second the link back to the source code is hard to establish. If, for example, a code coverage tool wants to do source visualizations, it not only needs to modify code but also link these modifications to the source code parts they belong to. These disadvantages are not specific to ByteSurgeon but the approach of using binary code to reflect upon method execution.

Consequences

This limited support for reflection at the sub-method level makes writing custom tools that deal with code or metaprogramming harder than necessary. It is hard because code is represented with objects that offer only a very low level of abstraction. Because of this the tools that manipulate code in Squeak internally use a custom, high level representation to to the actual work. But as there is no standard one, there is an uncontrolled growth of incompatible representations that all try to solve the same problems. Bytecodes and strings are used only for communication with other tools and to present code to the programmer because they are the only common model of code for tools in Squeak. Often a constant switch of abstraction levels between internal high level ones and external low level ones is required. If there was a standard high level representation of code, shared by all the tools, they could do all their work at the high level abstraction.

[2] `http://source.wiresong.ca/ob/`

As an example we want to build a tool that highlights all sends inlined by the compiler inside a method. Already the first step is difficult: finding out which sends are inlined. The advice given was to "Just have a look what the decompiler does.". This forces us to deal with a low level representation: bytecode. Once we have the required information we need to map it to another low level representation: text. Ideally whether a message send was inlined or not is a property on the message node. All that is left would be to tell the presentation engine to highlight these nodes.

Writing custom code tools or doing metaprogramming is not as uncommon as it might seem first. In fact it is quite common that students working at the SCG[3] find themselves forced to implement their own tracing tools because no general solution exists. This is undesirable because it has nothing to do with the actual research. It is therefore imperative to keep the development time of such tools as low as possible because all these projects have a limited time frame and this way every hour invested into building infrastructure for tools is not available for research. The students doing this come from every level starting at bachelors up to PhDs. These tools are used among others for tracing, object flow analysis [LDGN06], version analysis, Classboxes [BDW03], ChangeBoxes [NDGL06], experimenting with new language semantics. Currently a variety of tools is used including MethodWrappers [BFJR98], Objects-as-Methods [BD06], ByteSurgeon, or even changing the compiler.

[3] http://www.iam.unibe.ch/~scg/

Chapter 3

Related Work

Related work can be roughly categorized into three categories:

1. calls for a higher level abstraction of code that is rendered richer than just a text file
2. languages that provide reflection at the sub-method level
3. frameworks that simplify the creation of tools that manipulate with code

Higher Level Abstraction

Dimitriev [Dim04] argues that programs should no longer be text but a graph described with a metamodel built for a certain kind of problem. The language would be be mapped to another one for execution or interpretation.

Edwards [Edw05] argues that programs should no longer be text and the representation of a program should be the same as its execution. His programs are trees created by copying. He also identifies the need to customize the presentation of a program.

Black [BJ00] makes a case to free programs from their linear structure and replace them with a much richer abstract program structure (APS) that captures all of the semantics, but is independent of any syntax. Conventional one and two dimensional syntax, abstract syntax trees, class diagrams, and other common representations of a program are all different "views" on this rich abstraction.

Quitslund [Qui03] argues that programmers should be freed from the file centric view and given one that allows better juxtaposition of disjoint pieces

of code.

Fortess by Sun[1] aims at supporting mathematical notation. Two points mainly contribute to this. First it has full Unicode support for operators. Second it gives control over the rendering of an identifier by a naming convention. However, this is hard wired into the specification of the language.

ETMOP [EK06] allows annotations to control the rendering of AST nodes. *Render edit metaobjects* (REMOs) can even edit them which translates back to changed source code.

Sub-method Reflection

In LISP [McC60] source code is itself made up of lists. As a result macros can manipulate it using the list-processing functions available in the language. This functionality is limited to macros at compile time and can not be applied to functions at runtime. Listing 3.1[2] shows an example of a macro that implements a functionality similar to an incrementation operator.

Listing 3.1: Lisp Macro Exmaple

```
(defmacro inc (var)
   (list 'setq var (list '1+ var)))
```

In io [Dek05] code is a runtime inspectable and modifiable tree. Message arguments are passed as expressions and evaluated by the receiver. Selective evaluation of arguments can be used to implement control flow. Listing 3.2 shows an example where the expression `c + 1` is passed to the `#if` function. The expression can then be evaluated on need by the receiver.

Listing 3.2: io if Exmaple

```
if(b == 0, c + 1, d)
```

Slate[3] allows message sends to syntax nodes called *macro-level* message send. In this way the syntax tree can be modified or even made available at runtime. Preceding any selector with a back-tick (‘) will cause it to be sent to the parsed entity. Listing 3.3 shows an example where the message `#quote`

[1] http://research.sun.com/projects/plrg/fortress.pdf

[2] http://www.gnu.org/software/emacs/elisp-manual/html_node/Simple-Macro.html#Simple-Macro

[3] http://slate.tunes.org/

is sent to a message node which returns the node itself. Instead of `7` the code will evaluate to a binary message node.

Listing 3.3: Slate Exmaple

```
(3 + 4) 'quote
```

C# 3.0[4] will feature expression trees, efficient in-memory data representations of lambda expressions that make the structure of the expression transparent and explicit. The type `Expression<T>` can be used to indicate that an expression tree is desired for a given lambda expression rather than a traditional method body. Listing 3.4 shows an example where the variable `e` is a reference to an expression tree

Listing 3.4: Expression Tree Exmaple

```
Expression<Func<int, bool>> e = n => n < 5;
```

Frameworks

IDEs like Eclipse JDT[5] internally use an AST to represent code. It is more advanced than a traditional AST because it provides additional information and functionality that is needed for the services Eclipse JDT provides like refactoring. Code editing in such an IDE is no longer text editing but editing of nodes in a graph. The user interface completely hides this. Text and bytecode are just used for storage and execution. Besides all the advantages associated with the approach chosen by Eclipse it has several disadvantages. The first problem is that it is not portable to other IDEs. NetBeans[6] and IDEA[7] each use a different AST that offers the same functionality but is incompatible. This is very unfortunate for plugin writers because it unnecessarily increases their expense. If tool writers want to take advantage of the AST they have to build their tools as plugins for an IDE. They cannot make use of it in a stand-alone tool. Also metainformation that is emulated with comments (eg. a string is non-externalizable) cannot be shared by all IDEs. Because such a powerful AST is not part of the language, a long delay between the time features are added to the language and IDEs support them can result as with the Java 1.5 support in Eclipse. The second

[4] http://msdn.microsoft.com/data/ref/linq/default.aspx?pull=/library/en-us/dndotnet/html/linqprojectovw.asp#linqprojec_topic3

[5] http://www.eclipse.org/jdt/

[6] http://www.netbeans.org/

[7] http://www.jetbrains.com/idea/

problem is that the high level model of code is not persistent. For example it is not possible to directly check a conditional breakpoint into a version control system (VCS) and check it out on an other machine.

Initiatives like the Java IDE API[8] clearly show the need for a cross platform API to access and modify the source code in a programmatical and high level way. It is however questionable how widely this will be adopted by Java IDEs as the market is dominated by one product whose developers do not seem to be interested in implementing the specification.

APT[9] finds and executes annotation processors based on the annotations present in the set of specified source files being examined. The annotation processors use a set of reflective APIs to perform their processing of program annotations. The apt reflective APIs provide a build-time, source-based, read-only view of program structure.

[8] http://jcp.org/en/jsr/detail?id=198
[9] http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html

Chapter 4

Solution

In this chapter we first present a list of requirements for a reflective method. As the scope of the requirements presented is large, and due to the limited time we had, we focus on certain key areas while leaving out others as future work. We present the list of requirements we decided to postpone. Afterwards present our concrete model of a reflective method.

4.1 Requirements

A high level model is required to address the issues described in Chapter 2. We have identified several requirements for such a model.

Structure

First the model of code should capture the structure of a method. That is how expressions are nested, what subexpressions they contain and in what superexpressions they are. There should be a way to identify of what kind an expression is. For example if it is an assignment or a message send.

An entity should not be an array of entities with a type integer but an instance of a class that describes it. The model should be neither too simple nor too complicated. If the model is too simple it would just have one or two kinds of entities. Such a model would be too generic to be useful. If the model is too complex it would have a hundred or more kinds of entities. This makes working with such a model hard because there are always many different cases that must be handled.

Behavior

Just encoding the structure is not enough, the model should also include common behavior needed by tools so that a tool builder is not required to reimplement the same functionality in his code. This includes operations that change the structure, query the code, and methods to navigate through the model. We consider the behavior needed to implement refactorings a good measure for what behavior should be provided.

System Integration

The method should be the first class representation of code in the system. All tools like browsers, the compiler and the debugger must use it as their interface when working with code. This also includes the VM that should be able to work with such a method. The VM is allowed to transparently extend the method with a model that suits its needs better like bytecode as long as this is transparent for all the other tools including the debugger. A consequence of being the first class representation is that a method is causally connected, every change to it immediately takes effect.

Presentation Engine

When programmers interact with the tools they are most likely to do so by modifying this text view. Because of this we need to make it easy for tools to modify this view or build their own textual view. The following are presentational changes tools might want to make.

- **Different coloring**. A profiler draws a heat map on the code or a coverage tool colors executed code differently from not executed code. Another tool highlights all the inlined sends.
- **Adding errors or warnings**. This could for example be implemented by underlining code red or yellow. A pluggable type checker can use it for type mismatches. SLint can use it for rule violations.
- **Adding actions** allows tools to perform operations on model entities available via graphical interaction. This can include corrective actions for errors or warnings above. For example adding a `#yourself` to a cascade. It is also possible that this includes actions that cannot be done by editing source code.
- **Tooltips** can be used by tools to display additional information about code on demand. An object flow analysis tool can use it to display what objects were stored in some variable.

- **Customize the rendering** of certain entities. A plugin can use this to render certain parts of the code in mathematical notation.

This list is heavily inspired by the features of Eclipse JDT. Other sources of inspiration are Subtext and ETMOP (see Chapter 3).

Extensibility

All tools should use reflective methods as their first class model of methods, but it is not possible to anticipate the need of every tool. Thus it must be possible for tools to extend the model for their needs with both data and behavior. Otherwise the tools would again have to build their own model to support their data and behavior. This would mean a failure to achieve the most important goal, that all tools share the same model. A key requirement of the extension mechanism is that tools can make their extensions in way that does not conflict with extensions of other tools.

Storage

As the high level model of code is the first class abstraction for code in the system, it needs to be persistent. The code management tools need to be able to work with and store such a model. For the data extensions tools introduce there needs to be a mechanism to mark them as persistent or ephemeral.

Translation

The model should also support translation to other, especially low level, representations. This is needed for example to transform to text or bytecode. We cannot get rid of these two representations. Text is still the preferred way of how programmers perceive code and changing that is out of the scope of this thesis. The same way, bytecode is the preferred representation of code for VMs, although research indicates that other more efficient forms are possible [FK97]. This again is beyond the scope of this thesis.

Creation

The reverse way of creating the high level model from a low level abstraction must also be possible. Typing on a keyboard is still the preferred way of creating programs and changing that is out of the scope of this thesis.

Also there must be an easy programmatic way to assemble methods. Either directly for metaprogramming or indirectly via a visual programming tool.

Round Trips

When going from a low level representation to the high level model and back to the low level, there should be no information loss. In the case of text this means that as long as no changes in the high level model are made, the formatting must be kept. If the high level model is changed, as much low level information as possible must be preserved. For example renaming a variable or changing the selector of a message send should also preserve the formatting.

4.2 Consequences

Once we are at this point, text is no longer the main representation of code in the system. It only serves two purposes: one is presenting the code to the user, but this could also happen in another non-textual form. The other is the creation of the high level object that represents code. This too can also be done with another tool like a graphical editor.

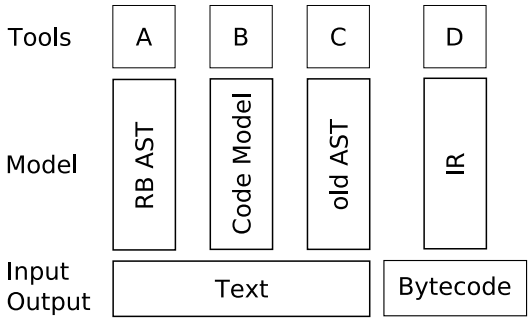

Figure 4.1: The Current Situation in Squeak

Figure 4.2 shows the current situation in Squeak with text and bytecode as first class representations. On top of that tools build their own, incompatible, high level models. Compared to that Figure 4.2 shows an ideal scenario with the reflective method as the first class representation of a method. Tools can directly use it as it already provides common behavior and can be extended. Lower level representations like text or bytecode are only generated on need.

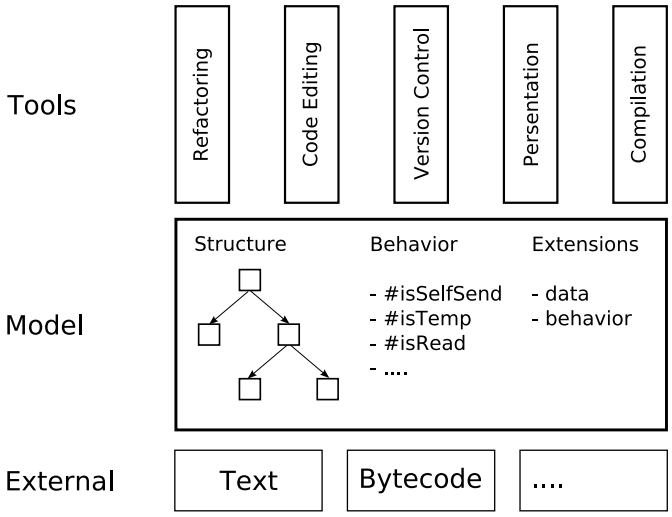

Figure 4.2: Integration of a Reflective Method in an Ideal System

4.3 Restrictions

As this project is limited in both time and resources we will allow ourselves the following shortcuts so that we can focus on what we consider the important parts that are achievable given these limits. Implementing any of the following would take away too many resources from other aspects.

- We do not have to keep formatting of code; instead we allow pretty printing of code. This is influenced by our choice of the underlying representation for the implementation that does not yet support this.
- We do not change any of the existing tools to work with this model of code. Instead we implement the current interface for methods in the system. In this way the tools do not need to be changed and can work with reflective methods as if they were `CompiledMethod`s.
- We do not build a presentation engine, we only update the pretty printer to work with our extensions.
- We also do not build storage support; instead we use translation to text and use the currently present code storage tools.
- The VM does not know about our model of code, so we allow swapping of our model with the old model on the first execution of the method. This way we do not have to change the VM and only have an initial runtime penalty.

4.4 The Model

Our implementation of reflective methods is called Persephone[1]. We took the RB AST as a starting point because it already provides a lot of the required functionality.

- **Structure** and **behavior** are already provided by the RB AST. We added additional behavior in the form of convenience methods.
- **Translation** to source code is provided by the pretty printer of the RB AST. Translation to bytecode is provided by the NewCompiler.

We added what was missing of the required functionality.

- For **extensions** with data a property mechanism was added. We call these properties annotations. An extension to the Smalltalk syntax allows annotations to be created from source code. Extensions with behavior can be done with class extensions.
- **Compiler plugins** provide a form of extensibility that cannot be reached with annotations alone. Annotations are static and do not directly affect the runtime behavior of a method. Compiler plugins transform a copy the AST that will be compiled to bytecode. In this way they can change the runtime behavior of a method without changing the original AST. An example of this a plugin that replaces message sends that result in a constant value with this value. Because it works on a copy, the original send is kept in the AST but removed from the bytecode.
- During the case studies we discovered that the RB AST lacked several features. We will present them and their implementation in Chapter 6 in the context of the related case study.
- Although we changed none of the exiting tools they still use reflective methods unconciously as their **first class** model of methods. Most tools use them through the interface of `CompiledMethod` which we implement. Tools of the RB tool stack directly access the AST through the method `#parseTree` which for instances of `CompiledMethod` parses the source code but in the case of reflective methods just returns the AST. Additionally we build several tools which directly use reflective methods as their first class model for methods.
- We implemented **causal connection** which is used by the tools we built and that are aware of reflective methods.

[1] After the greek goddess who spends half of her time in the underworld and the other half in the upper world.

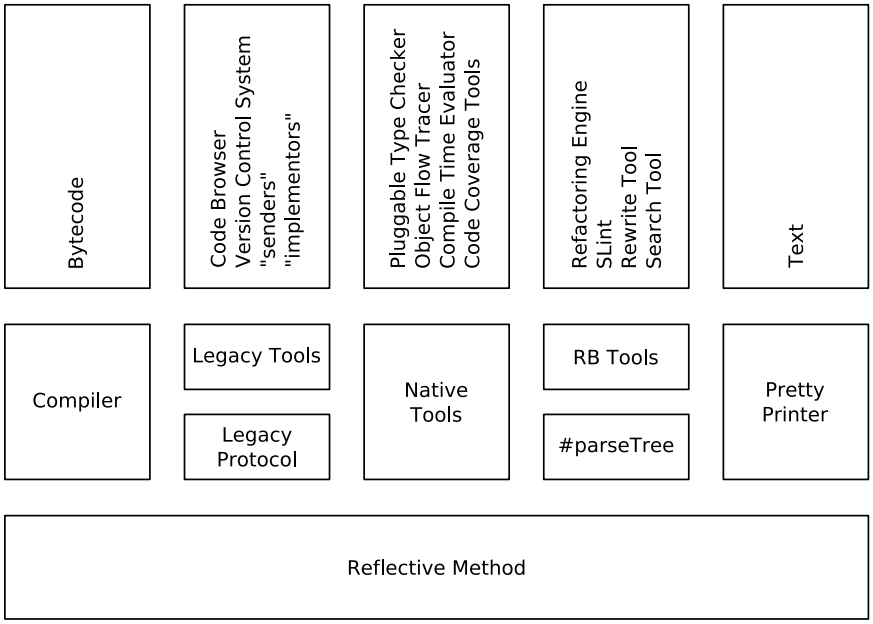

Figure 4.3: Persephone

Figure 4.3 outlines Persephone with the reflective method as the first class representation of a method that is causally connected to the system. The native tools we built in Chapter 6 directly use it. Text and bytecode are only artefacts created by the compiler and the pretty printer. For legacy tools we implemented the interface of `CompiledMethod`. Tools of the RB tool stack directly access it through `#parseTree` which returns the annotated AST instead of parsing the source code.

The details of the implementation and the system integration are presented in following chapter.

Chapter 5

Implementation

In this section we describe in detail our implementation of reflective methods. First we outline our choice of underlying technology. This choice is dominated by ease of use and effort required to be able to work with it. Then we show how we integrate our reflective methods into the existing Squeak system. After that we introduce annotations as our solution to extensibility. In the end we present the modifications we made to the compilation process to support our model.

5.1 Underlying Technology

Our implementation is based on the NewCompiler and RB AST. We choose the RB AST because it is an AST that provides a lot of the needed functionality and is used successfully in the whole RB tool stack and the NewCompiler. If we had not chosen the RB AST we would have to rebuild its entire functionality because the functionality RB AST provides is a key part of our requirements for reflective methods.

We chose the NewCompiler over the stock compiler for two reasons. First, whereas the old compiler uses its own AST the NewCompiler uses the RB AST as its model for code. In this way we can directly pass an RB AST to the compiler without transforming it to something else or changing the compiler. Second, the NewCompiler is built using a more modern, object oriented design. For example it uses the visitor pattern to walk over the AST. This makes it significantly simpler to understand and extend. We chose the parser generated by SmaCC [BR] instead of the hand written one of the Refactoring Browser. This makes experimenting with syntax extensions simple because only the parser definition has to be changed. However, it places also certain restrictions of these extensions by allowing LR(1) grammars at best.

Our choice has one significant downside however. Going from AST nodes to text can only be done by using the pretty printer of the Refactoring Browser. During this process all formatting information is lost. There are, however, other ASTs like CodeModel that preserve the formatting.

We believe this AST is a good way to represent code for several reasons. It works well for high level operations like refactorings. It is close to the mental model of the programmer because nodes in an AST are the building blocks of the language: message sends, assignments, blocks and so forth. Source code generation can be done using pretty printing. Given an advanced pretty printer like the one of Eclipse JDT this generated code is only slightly different than what the programmer typed. This can be further improved by implementations that preserve whitespace. And finally the AST also allows direct bytecode generation using the NewCompiler.

5.2 System Integration

To integrate our reflective methods we make use the way methods are stored and created in Squeak. In Squeak classes are fist class objects that are available to any program. They have an instance variable named `methodDict` which holds an instance of `MethodDictionary`—a special subclass of `Dictionary`. All methods of a class are stored in this method dictionary. The VM directly uses the class objects and their method dictionary when performing sends. Normally only instances of `CompiledMethod` are stored in the method dictionary of a class but Squeak supports to store any kind of object there. The VM recognizes objects that are not instances of `CompiledMethod` and instead of executing their bytecode the VM sends `#run:with:in:` to the object stored in the method dictionary. Methods are created by sending one of the `#compile:` methods with the source code as an argument to the class to which the method belongs. This will first compile the method to an instance of `CompiledMethod` and then add this method to the method dictionary. The compilation is delegated to the compiler which is found by sending `#compilerClass` to the metaclass. The method returned by the compiler is then added to the method dictionary by using one of the `#addSelector:` methods. Figure 5.1 visualizes this relationship between a class, the method dictionary and and the methods.

We implemented a custom compiler that creates reflective methods instead of instances of `CompiledMethod`. These methods implement the public interface of `CompiledMethod` so that they are indistinguishable for the existing, not altered tools. Besides the extended and annotated RB AST these methods also reference an instance of `CompiledMethod` which is generated on need. This is the case if the method is sent a message that cannot be

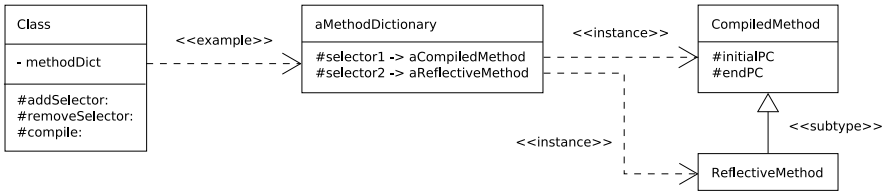

Figure 5.1: Class-MethodDictionary-Method-Relationship

asked by looking at the AST like `#endPC` or if the VM wants to execute this method. The VM can only execute instances of `CompiledMethod` so when it encounters a reflective method it sends `#run:with:in:` to the method. If this happens we replace the reflective method in the method dictionary of the class with the compiled method so that further message receives will no longer be slowed down. This relationship between reflective and compiled method is shown in Figure 5.2.

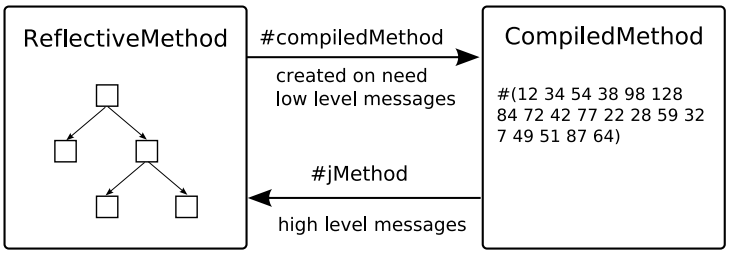

Figure 5.2: ReflectiveMethod to CompiledMethod Relationship

There are two different ways to get a class compiled with reflective methods. The first is to override `#compilerClass` on the class side and return our compiler: `JCompiler`. Second, we implemented a trait[1] named `TJMethod` that does this. The advantage of using any of these two approaches is that all methods in such a class or any of this subclasses will be compiled with our compiler. The disadvantage is that the class has to be changed and it does not work for methods on the class side. This makes it ill suited for case studies with existing code. Therefore we implemented a helper class named `JRecompiler` that can be used to recompile individual methods or whole classes with our modified compiler.

If a change to the reflective method is made, we discard the compiled method and move the reflective method back into the method dictionary of the class should it not already be there. In this way the compiled method acts like a

[1] Traits are fine-grained components that can be used to compose classes, while avoiding many of the problems of multiple inheritance and mixin-based approaches. For further information see `http://www.iam.unibe.ch/~scg/Research/Traits/index.html`

cache for bytecode that is invalidated when the reflective method is changed. Figure 5.3 illustrates the process of swapping methods in the method dictionary. First a reflective method is the method dictionary. Once the VM tries to execute it a compiled method is generated and takes the place of the reflective method in the method dictionary. Later when the reflective method is modified, the compiled method is discarded and replaced by reflective method until the VM tries again to execute the reflective method.

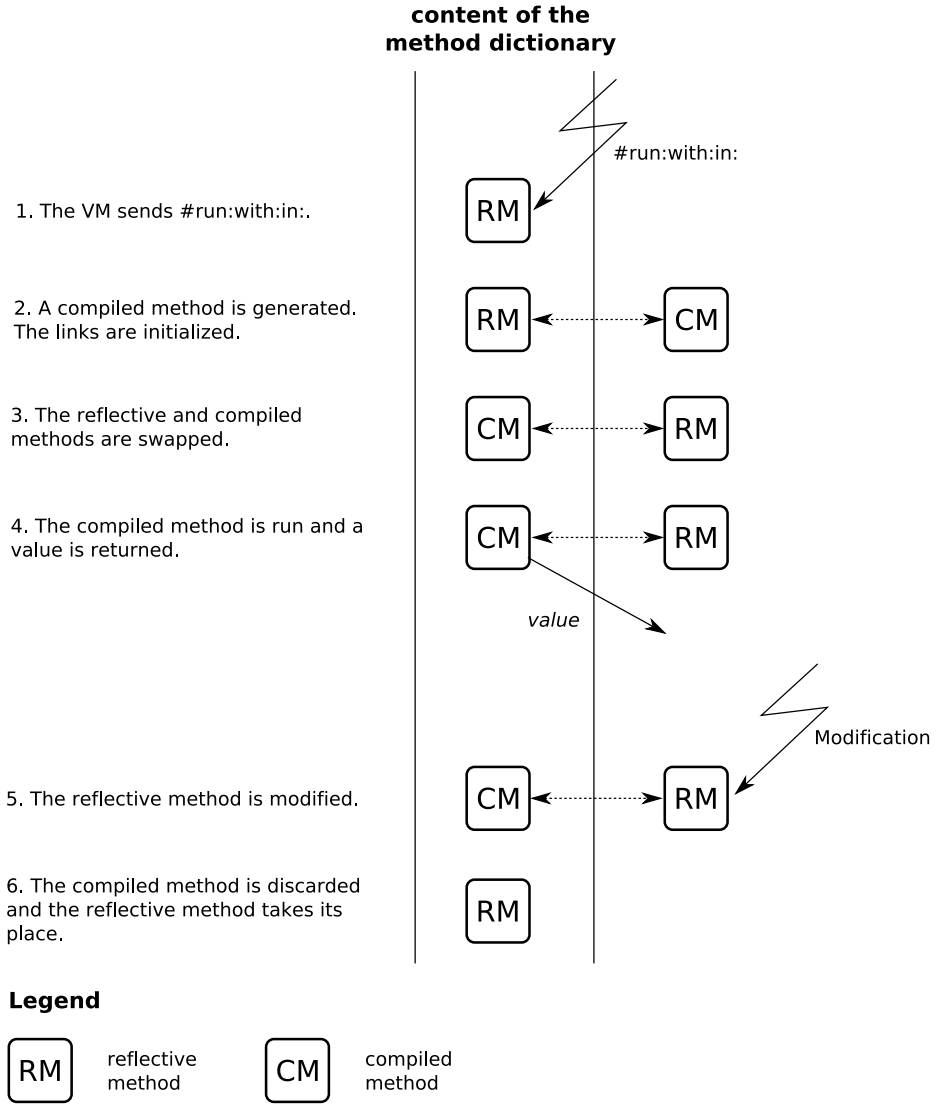

Figure 5.3: Contents of the Method Dictionary over Time

Compilation on demand can result in significant time savings compared to

ahead of time compilation. Real world examples ([HDD06]) have shown that in a typical feature invocation only 10% of the code of a package is executed. In this way all the methods that are not executed are only parsed and their compilation time is saved. This is an effect similar to a just in time compiler that generates native assembly only for methods that are often executed. As a rule of thumb, the bigger an application is the smaller is its relative working set. This gives the time savings by lazy compilation a leverage.

5.3 Annotations

As mandated in Section 4.1 we would like reflective methods to be used as the first class representation for all users. This means that users of reflective methods need to be able to add information about objects (nodes) directly to those objects. This allows them to communicate using this model. Extensions to behavior can be made using class extensions, but extensions with data require more work. In a standard domain class this would be done by adding instance variables to a subclass. However this would lead to conflicts as soon as two users subclass the same model class. It is no longer clear which one should be used and a user might get a different one than he expects.

There already exits an extension mechanism for methods in Squeak 3.9 called *method properties* that we took as an inspiration for our solution. In the following we will first present method properties together with some examples and show why for some applications the scope of method properties is too coarse. Then we will present annotations as our solution for extensibility. At the end of this section we will to into the details of using annotations.

Method Properties

Our solution is heavily inspired by the method properties introduced in Squeak 3.9. Method properties allow users to add data to methods. This way information about methods can be stored in methods. For this each method has a `MethodProperties` object that implements a very basic `Dictionary` protocol. On top of method properties pragmas are implemented. Pragmas are special method properties that have a representation in the source code of a method. They are set by the writer of the source code at compilation time instead of programmatically later. This also means they survive recompilation because they have a source representation and source code is used for storage in Squeak right now. They use the existing primitive syntax of a message send in angle brackets as shown in Listing 5.1.

Listing 5.1: Pragma Syntax

```
aMethodHeader
    <aSelector: 'anyLiteral'>
    "code follows here"
```

Method properties and therefore pragmas too work by definition only on methods. That does not mean there is no use for pragmas defined only on parts of a method and not a method as a whole. As an example there is an extended version of SLint[2] that gives the programmer the ability to exclude false positives with pragmas as shown in Listing 5.2.

Listing 5.2: SLint Pragma

```
defaultBackgroundColor
    <lint: #expect rule: #overridesSuper rational: 'we want a different
        color than the parent'>

    ↑Color orange.
```

For some applications the scope of a method is too coarse. SLint rules for example have different scopes. Some are defined on classes like "Has class instance variables but no initialize method" while others like "Overrides super method without calling it" are defined on methods. Even others like "missing yourself" are defined on cascades. Pragmas are defined at the method level, if we use them to suppress false positives of SLint rules defined on sub-method structures like cascades this may result in hiding true positives. A method can contain several cascades and it is possible that whereas one of them is a false positive an other is a true positive. A pragma would hide them both because pragmas are defined at the method level. Another example for a property on a sub method structure would be the ability to mark certain string literals as non-externalizable.

Overview of Annotations

We take the idea of method properties to the the level of the AST node. Instead of properties we call the data extensions annotations whether or not they have a source representation. Each node has a dictionary that maps symbols to annotations. Annotations are instances of a subclass of `Annotation` that can have zero, one or multiple values. Our first experi-

[2] http://mc.lukas-renggli.ch/essential/

ments used simple key value mappings but this caused several problems due to the the variety of different annotations.

Annotations consist of a key (a symbol) and optionally one or multiple values. There exists three different classes that can be subclassed in order to create a custom annotation class. Instances of a subclass of `NoValueAnnotation` are for annotations that have no value. They are either present or not. Annotations with only a single value are subclasses of `SingleValuedAnnotation` whereas annotations with multiple values are subclasses of `MultiValuedAnnotation`. Figure 5.4 shows a diagram of the annotion classes.

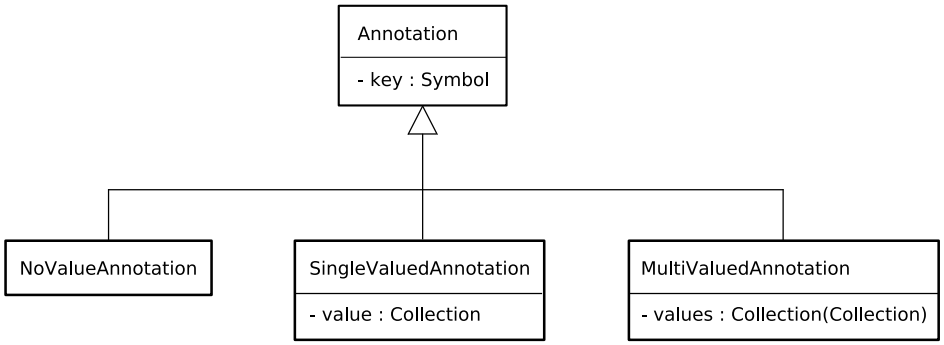

Figure 5.4: Annotation Hierarchy

The difference between multivalued and single valued annotations is that the former can be defined multiple times on the same expression with different values whereas the latter cannot. Examples of multivalued annotations can be found in Section 6.2.

In the same way that pragmas are method annotations that are created from source code we also need annotations on nodes that have a source representation so that they an be created by a programmer when writing code. Ideally we would have liked to use the pragma syntax: a message send without receiver in angle brackets. Unfortunately this is not possible in an unequivocal way that works with SmaCC. So we added additional colons. Only unary and keyword message sends are supported and not binary sends. A statement with an annotation now looks like in Listing 5.3.

Listing 5.3: Basic Annotation Syntax for Statements

```
aStatement <: aSelector: anArgument :>
```

This is supported on all statements and additionally on method arguments, block arguments and each variable name in a temporary variable definition.

Annotations with an unary selector result an a `NoValueAnnotation` since they do not have any argument. Annotations with keyword selector result in either a `SingleValuedAnnotation` or a `MultiValuedAnnotation`. This is independent of how many arguments the selector has. The value of a single valued annotation is a collection of its arguments. In the case of a single argument it is a collection with only one element. Multivalued annotations can be defined multiple times with different arguments on the same statement. Their value is a collection of the values of each definition.

Using Annotations

When the parser creates an annotation it searches all subclasses of `Annotation` for a specific class for this selector. If none is found, a generic annotation is created. In order for the parser to find the correct annotation class for a selector, `Annotation` subclasses have to implement the `#keys` method on the class side which returns the collection of selectors for this annotation.

Annotations may or may not appear in the source code. To control their visibility they have to implement `#isSourceVisible` on the instance side which returns a boolean. If it returns `true`, then the annotation and its value will be printed by the pretty printer.

The values of pragmas are restricted to literals. They can only be numbers, strings, symbols, booleans and `nil`. For annotations on statements in the source code, we removed that restriction and allow any value expression similar to the arguments of a message send. As an example Listing 5.4 is an invalid pragma because the value of a pragma must not be a message send. However Listing 5.5 is a valid annotation because message sends are allowed as arguments of annotations. By default, if an annotation in the source code as an argument other than a literal, its value will be the AST nodes representing that argument. If the argument expression shall be evaluated at compile time the annotation class has to implemented `#evaluateAtCompiletime` and return `true`. An example usage for this is shown in Section 6.5 where it can be used to create type objects at compile time.

Listing 5.4: Invalid Pragma

```
aMethod
    <belongsTo: self class name>
    ↑self
```

Listing 5.5: Valid Annotation

```
aMethod
    ↑self <:belongsTo: self class name :>
```

5.4 Compilation

Compilation is a multistaged process shown in Figure 5.5. Additions done by Persephone done to the compilation process are in boxes with broken lines. Starting from the annotated AST the compilation is handed over to a compilation strategy object. After that all compiler plugins are executed on a copy of the AST. Finally the traditional NewCompiler compilation is started.

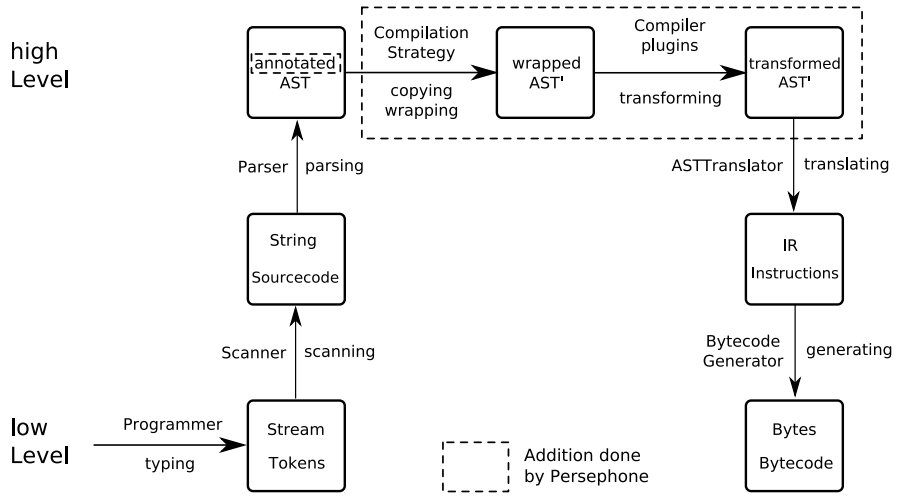

Figure 5.5: The Compilation Process

The compilation strategy is an addition to the compilation process we made. It is the starting point of the compilation and determines how a method is compiled. The two existing compilation strategies are used to implement method level instrumentations and thus are closely related to each other, they even have a common superclass to share code. The compilation strategy can be set by sending `#compilationStrategy:` to the method with the strategy class as an argument. The two existing compilation strategies are compared in Table 5.1 and presented below.

`JWrapperCompilationStrategy` compiles a method to a structure similar to a MethodWrapper: an "outer" method contains the instrumented code at the method level and an "inner" method contains the original method. This is simpler to implement modifying a copy of the method body but comes

with a performance penalty because two methods are actually executed even if the lookup only happens for one. One big advantage is that it allows primitive methods to be instrumented at method level.

`JInlineCompilationStrategy` inserts the instrumented code at the method level into the original method body. This is harder to implement because a copy of the existing method body has to be modified. However it creates more efficient code because one method activation is saved.

	Inline	**Wrapper**
Default	yes	no
Overhead	low	higher, like MethodWrappers
Primitives	no	yes
Implementation	harder	easy

Table 5.1: Inline versus Wrapper Compilation Strategy

After that both strategies run all the compiler plugins over a copy of the method node. A compiler plugin is just a subclass of `RBProgramNodeVisitor` that answers `#isCompilerBackendPlugin` with `true`. The return value of `#priority` is used to sort plugins in a deterministic order. In this way it can be controlled which plugins are run before others. Plugins affect the compilation by transforming the AST. This can happen in interaction with an annotation. Examples can be found in Section 6.1.

Finally the compilation using the NewCompiler happens. First a visitor walks over the AST and translates the nodes to IR. Then a second visitor walks over this and generates bytecodes.

Chapter 6

Validation

In this section we validate the claims that our implementation of reflective methods will ease metaprogramming at a sub-method level and the creation of tools. First we show how compiler plugins can be used together with annotations to build tools that change the compilation of a method. Then we present ByteNurse, a code transformation tool, which is used in the following case studies. Section 6.3 makes use of advanced instrumentation facilities of ByteNurse and shows how easily complicated instrumentations can be built. Section 6.4 shows an example of annotations used to store information about nodes and makes use of the node exposed to instrumentations and the AST as a first class representation of code for tools. Section 6.5 shows a tool that is built on annotations on expressions and the AST as a first class representation of code. Where possible we added benchmarks at the end of the section to assess the achieved performance.

6.1 Compiler Plugins

In this section we present two compiler plugins that were built only within the context of this project. In the following section we present a larger compiler plugin that is used in the later sections.

Conversion to JavaScript

Modern web applications consist not only of HTML[1] but also of JavaScript[2]. Often not the whole JavaScript is written but a library is used like in other parts of application development. There is however still the need for glue

[1] http://www.w3.org/MarkUp/

[2] http://www.ecma-international.org/publications/standards/Ecma-262.htm

code between the JavaScript library and the application. This code has to be JavaScript. Seaside[3] provides an API for generation of HTML with plain Smalltalk code instead of templates. This has the advantage that all the Smalltalk code tools work on this code. For JavaScript there is also an API that allows the generation of JavaScript code from Smalltalk code at runtime[4].

For example the Smalltalk code shown in Listing 6.1 will create the Smalltalk string of JavaScript code shown in Listing 6.2.

Listing 6.1: Original Smalltalk Code

```
SUEffect new
    id: 'bar';
    duration: 2;
    shake
```

Listing 6.2: Generated JavaScript Literal

```
'new Effect.Shake(''bar'',{duration:2})'
```

Without tool support this can use up computation time to the point where it negatively impacts the performance of the system. We developed a compiler plugin that performs the transformation from Smalltalk code to JavaScript at compile time. For that to happen an expression has to be annotated as in Listing 6.3. The generated bytecode will be equivalent to the source code in Listing 6.4.

Listing 6.3: Annotated Smalltalk Code

```
anEffect
    ↑(SUEffect new
        id: 'bar';
        duration: 2;
        shake) <: asJavascript :>
```

The `asJavascript` annotation tells the compiler plugin that this expression should be converted to JavaScript. The plugin then runs the conversion code and replaces the expression which the resulting string.

[3] http://www.seaside.st/

[4] http://www.esug.org/conferences/14thinternationalsmalltalkconference2006/conferenceprogram/web2.0forseaside/

Listing 6.4: Source Equivalent of the Genereated Bytecode

```
anEffect
    ↑'new Effect.Shake(''bar'',{duration:2})'
```

To perform the required transformations at compile time we need the ability to evaluate parts of an AST. In this case a cascaded message send with literals as arguments to a global variable(a class). We solved this by implementing a method `#evaluate` in `RBProgrammNode` that packs the current node into a DoIt node, compiles and then evaluates it. It is overridden in some subclasses so that it works on all the code that does not need to be bound to a receiver.

The whole implementation of the JavaScript plugin could be done without changing any of the existing classes in the system. Only two classes were added, the compiler plugin and the annotation.

The annotation class is named `FJCompiletimeAnnotation` and a subclass of `NoValueAnnotation` because its key is an unary selector. It consists of only two methods on the class side. The convenience accessor method `#key` which returns `#asJavascript` and the `#key` method which returns an array with only one element, the return value of `self key`. This means `#asJavascript` is the only selector which we can use to create instances of `FJCompiletimeAnnotation`.

The compiler plugin class is named `FJCompiletimeEvaluator` and a subclass of `RBProgramNodeVisitor`. On the class side it implements `#isCompilerBackendPlugin` and returns `true`. This marks it as a plugin. It also implements `#priority` which returns an integer to allow ordering. Besides that it only has two small methods on the instance side.

Listing 6.5: FJCompiletimeEvaluator >> #visitNode:

```
visitNode: aNode
    ↑(aNode hasAnnotation: FJCompiletimeAnnotation key)
        ifTrue: [ self evaluateNow: aNode ]
        ifFalse: [ super visitNode: aNode ]
```

Listing 6.5 checks every node if it has the `asJavascript` annotation. If this is the case then it will pass it to `#evaluateNow:` with does the conversion to a JavaScript string. If that is not the case it just continues the visiting process.

Listing 6.6 creates a message node that sends `#asJavascript` to a copy of the annotated node. This send is then performed at compile time by

Listing 6.6: FJCompiletimeEvaluator >> #evaluateNow:

```
evaluateNow: aNode
    | value literalNode |
    value ←(RBMessageNode
        receiver: aNode copy
        selector: #asJavascript) evaluate.
    literalNode ←RBLiteralNode value: value.
    aNode replaceWith: literalNode.
    ↑self visitNode: literalNode
```

sending it `#evaluate`. The return value will be a string so it can be placed in a literal. A literal node is created to hold it. Then the original node is replaced by this literal and the visit process is resumed there.

It is not necessary to always create a specific annotation class, the example would also work with a generic annotation. The fallback is graceful. If `FJCompiletimeEvaluator` is not present in the system or deactivated no transformation is made but the annotation is still added. If the annotation class is not present a generic annotation will be created that is ignored by other plugins. This will leave the code unchanged as if no annotation had been made.

The transformation works at the expression level, not the method or even class level. This gives the developer very fine grained control over what should be converted. This is needed because conversion at compile time is not possible if runtime information is to be put into the generated JavaScript code. However we could also make sure that only literals and no variables are referenced. An example of such analysis is shown in the following.

Evaluation at Compile Time

To investigate evaluation at compile time we built three different plugins. One that requires explicit developer action and two automatic ones.

The first one is based on annotations. Expressions have to be marked with an annotation for evaluation at compile time. Any kind of object is supported as a result and will be stored in the literal frame of the method. The mechanism described in Chapter 5 is used to carry to the evaluation.

The two other plugins work automatically and do not need an annotation. They search for message sends that include only literals. If they find one, they perform it using the very same mechanism as described above. The difference between them is that one will accept only literal values as return

Listing 6.7: Evalute at Compiletime Annotation

```
aConstant
    ↑(9 raisedTo: 9) <:evaluteAtCompiletime :>
```

values whereas the other will accept any object. The one that accepts only literal values as return values might end up doing one send too much which will force it discard the result and go back to the previous result. This can not be prevented because there is no way of telling what kind the result of a message send in Smalltalk will be.

Having any kind of object as literal is already supported by Squeak. However the RB AST allowed only Smalltalk literal objects as literals. We implemented the class `JObjectLiteralNode` that allows to put any object into a literal. Compared to traditional literal nodes it does not try to create a token for its value. This has the drawback that there is no textual representation possible, therefore it is intended to be used only by plugins.

Conclusion

The presented examples have several issues. Debugging does not work as expected, it breaks "senders" in the browser, the programmer does not get informed about the optimizations and last but not least the code does not get updated once the implementation of methods sent at compile time changes. These issues are all addressed in Chapter 7. Due to these issues we think that the presented plugins are suited only for a release compiler that optimizes code once it is debugged. They can be selectively switched on and off individually so that development can happen without optimizations taking place.

Despite these issues compiler plugins turned out to be an easy way to hook into the compilation progress. They can influence the compilation process on a high level of abstraction. If more control is needed in can be achieved by implementing custom node classes and adding class extensions to the compiler. Custom optimizations could be implemented with only very little code and no change of the syntax. Compiler plugins work well together with annotations and the RB AST. They especially profit from the convenience methods we implemented since they programmatically modify the AST. ByteNurse—the code transformation tool presented in the following section— is implemented using compiler plugins as well.

6.2 ByteNurse

In this section we present ByteNurse a code transformation tool similar to ByteSurgeon [DDT06]. To make the description more clear this section will be interwoven with examples. Compared to ByteSurgeon and other code transformation tools ByteNurse has the following unique properties:

- It works on AST nodes and not binary code. This not only raises the abstraction level but also it makes links to source code simpler because the AST node is directly available.
- The transformation is stored as an annotation on the AST node. This way it keeps track of what changes are done where and the original AST is left untouched.
- Code transformations can be expressed in a blocks of Smalltalk code instead of specially formatted strings.
- The generation of the transformed is done lazily, on need. Transformation merely results in the bytecode cache to be reset. This can result in considerable time savings especially for large case studies.

Listing 6.8: A First Instrumentation Example

```
method ←(SomeClass >> #someSelector) jMethod.

method instrument: [ :each |
    each isAssignment ifTrue: [
        node replace: [ :variable :value |
            variable ←value + 30000 ] ] ].
```

Listing 6.9: A Concreate Transformation done by Listing 6.8

```
a ←1 max: 3 ⟹ a ←(1 max: 3) + 3000
```

Listing 6.8 shows a first example that results in a concrete transformation shown in Listing 6.9. It replaces all assignments in SomeClass >> #someSelector with new one that additionally adds 3000 to the value to be assigned to the variable. First we need to grab the high level method object. We do this by accessing the method stored in the method dictionary of `SomeClass` at `#someSelector`. This might either be a compiled method or a reflective method so we send `#jMethod` to be sure to have the latter. We instrument the method sending `#instrument:` with a block as an argument. All the instrumentation methods have to be sent inside the `#instrument:` block. This is an iterator over all the nodes in the AST that also makes sure the

compiled method is reset. We only want to instrument assignments so we select them by sending `#isAssignment` to each node passed to the block. For the most common node types there are convenience methods to instrument only these. In the case of assignments it is called `#instrumentAssignments:` and shown in Listing 6.10.

Listing 6.10: Instrument Only Assignments

```
method instrumentAssignments: [ :each |
        each replace: [ :variable :value |
                variable ←value + 30000 ] ].
```

We then replace each assignment with an other assignment that adds `3000` to the original value. We do this by sending `#replace:` to the node and pass an instrumentation block as an argument. Additionally there exist the methods `#insertBefore:` and `#insertAfter:` to add code before or after a node and take an instrumentation block as well. The instrumentation block is a static description of the code to be inserted. It is not used directly but instead is decompiled and its body is stored in a multi valued annotation of the instrumented node. However, decompilation of blocks is not yet fully implemented in the NewCompiler. We use a combination of a pragma and a class trait to make sure that a method is compiled with the old compiler. The trait implements `#addSelectorSilently:withMethod:` which looks for methods with the pragma `<needsOldCompiler>`. If such a method gets added and the `#compileUseNewCompiler` preference is set, this means it was compiled with the NewCompiler. In this case the trait recompiles the method with the old compiler before adding it to the class.

During the transformation phase of the compilation, a plugin will search for nodes with an instrumentation annotation. If it finds one it will replace the node with an instance of a special subclass that can have code before and after it. During the translation phase, class extension methods in the translator know about instances of these classes and insert the code at the right place. Unfortunately it was not possible for us to reuse the existing translation methods and we had to copy and paste their bodies and then extend them.

Because the instrumentation block is not used directly but decompiled the only variables that can be directly referenced from inside the block are `self`, `super` and `thisContext`. They will be bound to their values at runtime not at instrumentation time. For everything else block arguments have to be used. Each kind of node provides a different set of metavariables that can be used as block arguments. An overview of all available metavariables can be found in Table 6.1.

Node	Metavaraible	Description
any node	`node`	the node itself
message node	`receiver`	the receiver of the message
	`arguments`	a collection of all arguments
	`firstArgument` up to `fifteenthArgument`	the argument at that index
	`argument`	the first argument
	`lastArgument`	the last argument
	`selector`	the selector of the message
method node	`arguments`	a collection of all arguments
	`firstArgument` up to `fifteenthArgument`	the argument at that index
	`argument`	the first argument
	`lastArgument`	the last argument
	`selector`	the selector of the method
assignment node	`variable`	the variable to be assigned a new value
	`variableName`	the name of the variable
	`value`	the value to be assigned to the variable
return node	`value`	the value to be returned
variable node	`value`	the value of the variable
literal node	`value`	the value of the literal

Table 6.1: Supported Metavariables on Nodes

Only one metavarible is supported on all nodes. It is the metavariable named `node` that is a reference to the original node of the statement beeing instrumented in the untransformed AST. This is interesting for any tool that needs a connection between runtime and code. Most often these are tools that collect information at runtime and then present this in the code in some ways. Examples are profilers that draw a heat map on the code. It could also be used to build macrosystems, especially when combined with special annotations.

Additional metavariables can be added easily. The `#metaVariables` method in the corresponding node class has to be implemented or changed. It returns a Dictionary that maps metavariable names to the selectors of the methods that return them. An important thing to remember is that all the metavaribles must return AST nodes. In order to simplify the process we implemented `#asLiteralNode` in `Object` which returns an `RBLiteralNode` wrapping the receiver. If the receiver is not a Smalltalk literal object a

`JObjectLiteralNode` is created instead of the normal `RBLiteralNode`.

If an other variable or otherwise computed value should be used inside an instrumentation block it has to be injected by adding the `#using:` selector part. It takes an association or collection of associations as argument that maps symbols to values. The symbols can then be used as arguments in the instrumentation block. In Listing 6.11 `increment` is bound to the value of `self incrementationValue`. It is added as an argument to the instrumentation block and then used inside it.

Listing 6.11: Use `#using:` to Inject a Variable

```
method instrumentAssignments: [ :each |
    each
        replace: [ :variable :value :increment |
            variable ←value + increment ]
        using: #increment −> self incrementationValue ].
```

Sometimes the behavior of only one object is to be changed. This is done by using the `#instrument:for:` method instead of `#instrument:`. The second argument is the object whose behavior will be changed. Behind the scenes, a new anonymous class is created for this object to which the instrumented method is added. This has the advantage that at runtime no checks need to be made to see if the method is executed for this special object.

Methods themselves can also be instrumented. Code can be inserted either before or after the normal method execution. This can be done using the `#addAfter:` and `#addBefore:` methods that take an instrumentation block like their siblings for sub-method nodes. There are also versions that take an additional `#using:` argument to inject values into the instrumentation block. Semantics for the before and after code are the same as for MethodWrappers and shown in Listing 6.12.

Listing 6.12: Before and After Method Semantics

```
anInstrumentedMethod
    self beforeCode.
    ↑[ self normalCode ]
        ensure: [ self afterCode ]
```

Listing 6.13 instruments a method so that every time it is executed it notifies a trace tool. Sending `#executionCollector` would return some object interested in what methods were executed with what arguments. By making the example a little bit more generic it would already provide all the information required by a full blown tracer.

Listing 6.13: Instrumentation for a Basic Tracer

```
method ←(SomeClass >> #someSelector:) jMethod.

method
    addBefore: [ :collector :reference :arguments |
        collector methodExecuted: reference withArguments: arguments ]
    using: {
        #reference −> (MethodReference
            class: SomeClass
            selector: #someSelector:).
        #collector −> self executionCollector } ]
```

Further examples and usages can be found in the following two sections. At the end of each we will conclude how ByteNurse worked in this context. A general conclusion about ByteNurse can be found in the following Chapter 8 as it builds on the conclusion of the individual case studies.

Benchmarks

In the following we will investigate the performance penalties of instrumented code. We focus on the efficiency of the generated code. Therefore the instrumented code does only very little work so that the instrumentation penalty is more obvious. The original code is shown in Listing 6.14.

Listing 6.14: Uninstrumented code

```
action
        ↑6 ∗ 9
```

Listing 6.15: JCounter >> #increment

```
increment
        count ←count + 1
```

First we only look at the performance of the resulting code when code is added before a method. So before executing the original code `JCounter increment` as shown in Listing 6.15 is inserted which increments a class instance variable.

- *Hand-coded* is the time for running a modified, hand-crafted method shown in Listing 6.16 where the additional code was inserted at the

Listing 6.16: hand-crafted before code

```
action
        JCounter increment.
        ↑6 * 9
```

beginning of the method body. This is equivalent to optimal performance and no overhead.

- *ByteNurse* is the time for running an instrumented method generated by the instrumentation framework presented in Section 6.2.
- *MethodWrappers* is the time for running an instrumented method generated using MethodWrappers.

Name	**Time**	**Factor**
Hand-coded	789	1
ByteNurse	787	1
MethodWrappers	10 416	13

Table 6.2: Code Added Before a Method

Table 6.2 shows that *ByteNurse* is about as fast as *Hand-coded* and more than ten times faster than *MethodWrappers*. This means the generated code is as efficient as hand written one and the overhead for adding code is about zero. The only overhead is the added code itself.

MethodWrappers show considerable overhead. This was to be expected since they actually generate two methods. In real world examples the relative overhead of MethodWrappers is likely to be much lower because methods in general do more work than just multiplying two numbers.

To assess the performance of code added at the end of a method we extended the benchmark above to send `JCounter decrement` as shown in Listing 6.17 before exiting from the method. This code is inside an `#ensure:` block to make sure it is always executed. This caused the value of *Hand-coded* to increase considerable compared to the benchmark above even though the code inserted after is the same as the one inserted before except that it does a decrementation instead of an incrementation of a value.

Listing 6.17: JCounter >> #decrement

```
decrement
        count ←count − 1
```

Listing 6.18: hand-crafted before and after code

```
action
        JCounter increment.
        ↑[ 6 * 9 ]
                ensure: [ JCounter decrement ]
```

Name	Time	Factor
Hand-coded	4 162	1
ByteNurse	4 267	1
MethodWrappers	11 047	2.7

Table 6.3: Code Added Before and After a Method

Table 6.3 again shows that *ByteNurse* is about as fast as *Hand-coded* and has almost no overhead. However the margin *ByteNurse* has over *Method-Wrappers* has shrunk and *ByteNurse* is now between two and three times faster than *MethodWrappers*. The absolute time of *MethodWrappers* has increased less than *Hand-coded* and *ByteNurse* have increased in absolute time. One possible explanation for this behavior is that MethodWrappers always produce an `#ensure:` even if it is empty.

6.3 Object Flow

In this subsection we discuss in detail the instrumentation part of implementation of 'Capturing How Objects Flow At Runtime' [LDGN06] that was done using ByteNurse. The paper argues that today's dynamic analysis approaches that are based on method traces do not cover all aspects of object oriented systems because the behavior of a program depends on the sharing and the transfer of object references (aliasing). It proposes a new approach that complements existing ones. This approach captures the life cycle of objects by explicitly taking into account object aliasing and how aliases propagate during the execution of the program.

This posed some unique challenges on the instrumentation needed to gather the flow information. Early prototypes used a modified, standard Squeak compiler. They showed that considerable effort would be required to implement all the required instrumentations.

- All assignments have to be instrumented. Instead of their real value an alias is stored in the variable.
- All instance variable reads have to be instrumented to reconstruct the

state of objects in the past.

- Inlining has to be deactivated in order to be able to trace `true`, `false` and `nil`.

In order to be able to trace the flow of objects an alias instead of the original value is stored in a variable. An alias captures what type of variable it belongs to (instance or temporary), the name of the variable and the old value.

Listing 6.19: Instrumentation of Assignments

```
method instrumentAssignments: [ :each |
    each variable
        ifTemp: [
            each replace: [ :variable :variableName :value |
                variable ←value
                    asTempAliasNamed: variableName
                    in: CurrentActivation value
                    predecessor: variable ] ]
        ifInstance: [
            each replace: [ :variable :variableName :value |
                variable ←value
                    asFieldAliasNamed: variableName
                    in: CurrentActivation value
                    predecessor: variable ] ]
        ifGlobal: [ "ignore" ] ].
```

In Listing 6.19 different code is generated depending to what kind of variable a value is assigned. This is done with the `#ifTemp:ifInstance:ifGlobal:` method, this is one the convenience methods we built into the AST. The last block is empty which means assignments to global variables will be left untouched. The assignment is replaced with an assignment that assigns a different value to the same variable. `CurrentActivation value` is a dynamic variable that represents the current method activation. This was implemented using MethodWrappers because ByteNurse at that time did not yet provide the required functionality.

The instance variable reads have to be instrumented in order to be able to create a view of the system in the past.

Only reads of instance variables are instrumented. In Listing 6.20 the corresponding nodes are selected using the `#isInstance` and `#isRead` testing methods. Although `name` is currently not used it is planned that future version a make use of it.

Listing 6.20: Instrumentation of Instance Variable Reads

```
method instrumentVariables: [ :each |
    (each isInstance and: [ each isRead ]) ifTrue: [
        each replace: [ :value :name |
            (Processor activeProcess backInTimeId isNil
                or: [ value isAlias not ])
                    ifTrue: [ value ]
                    ifFalse: [ value
                        xxxBackAt: Processor activeProcess backInTimeId ] ] ] ].
```

The flow of all objects is intended to be traced including the special objects `true`, `false` and `nil`. However this causes a problem because the compiler inlines certain—mostly control flow related—messages. The resulting code will not work any more with aliases. Thus inlining has to be completely disabled.

Listing 6.21: Deactivation of all Inlining

```
method instrumentMessages: [ :each |
    each isInline ifTrue: [ each doNotInline ] ].
```

The code in Listing 6.21 is straightforward and intention revealing. `#isInline` is a message already provided by the NewCompiler. If it returns `true` the compiler would normally do some kind of inlining. `#doNotInline` sets an annotation on the node. When processing it, the compiler will recognize it and not inline it.

The no inline annotation shown above is programatically set and does not appear in the source code. There are also no inline annotations that are set by the programmer in the source code. Such annotations can be used in an application that implements ternary logic. In such a program, besides the boolean values `true` and `false` a third value `unknown` exists which is neither `true` nor `false`. This too is incompatible with the standard inlining of the compiler which thus has to be deactivated. But in this case it is best done by the application programmer in the source code with an annotation because he knows where ternary boolean values are allowed and where not.

In future work, the assignment node for which an alias is created might be included in the alias as well. This would give the ability to do source visualizations and visualize the flow of objects in the source code of a method. A presentation engine for code as described in Section 4.1 would be simplify this task considerably.

Conclusion

The author appreciated the high level of abstraction ByteNurse offered. It allowed him to express the instrumentations in an intuitive and intention revealing way. On the negative side were some stability problems in early versions that were corrected during the usage.

In the case study used—the NewCompiler itself—the instrumented code was ten times slower than the uninstrumented code. Considering the amount of instrumentation done, that proxy objects were used and especially that control flow was done using message sends, we are satisfied with the performance.

6.4 Code Coverage

Code coverage analysis per expression is a conceptually simple task. Before an expression gets executed it is marked as executed. After the program is run the executed expressions are printed differently from the ones that were not executed.

Practically however, it is unnecessarily complicated because different levels of abstractions are involved. There is the conceptual level, at which we deal with expressions and how often they are executed. Besides that there is the actual level at which we compute the coverage of the expressions. The more these levels diverge the harder the task becomes. For traditional implementations the actual level is bytecodes and bytecode manipulation. This requires us to almost constantly switch abstraction levels from very high ones to very low ones. Right from the start we need to go from a high level of abstraction to a low one: we need to map expressions to bytecodes and instrument them. After running the code the expressions have to be extracted. Once we are on a high level of abstraction again and have the expressions we need to produce an other low level format, namely text. In contrast our model allows the tool builder to constantly work at a high abstraction level lifting the actual model to the level of the AST node. This makes the actual task almost the same as the conceptual task.

As shown in Section 5.3 the most convenient way store information about a node is by adding an annotation to it that hold the information. To to keep track of how many times a node has been executed we create a subclass of `SingleValuedAnnotation` named `CLExecutedAnnotation`. We then add a method `#markExecuted` to `RBProgrammNode` via a class extension as shown in Listing 6.22. If we now send `#markExecuted` to any node, its execution count will be incremented.

Listing 6.22: RBProgrammNode >> #markExecuted

```
markExecuted
    (self annotationAt: CLExecutedAnnotation key) increment
```

When we started with the code coverage case study our code transformation tool (ByteNurse) did not yet offer the ability to send messages to nodes. For prototyping and easy customization of the execution we built an interpreter as a visitor over the RB AST. This interpreter has support for primitives. It cannot however be interpreted itself because exception handling in Squeak requires native blocks. It can be switched on per method by sending `#beInterpreted`. A custom interpreter can be set by sending `#interpreterClass:` and the custom interpreter class as an argument. Benchmarks for this interpreter can be found at the end of this section. Our first implementation used a subclass of this interpreter. As it is a visitor over the AST it could be implemented by overriding only `#visitNode:` as in Listing 6.23.

Listing 6.23: Overriden `#visitNode:`

```
visitNode: aNode
    aNode markExecuted.
    ↑super visitNode: aNode
```

Although this is slow in relative numbers compared to execution by the VM it was still fast enough to run most case studies. This allowed us to very rapidly build a prototype while the instrumentation interface was not yet fully working.

Our second revision uses instrumentation and the `node` metavariable to mark nodes as executed shown in Listing 6.24. The node is stored in the literal frame of the method by making use of the RB AST extension described section that allows to store any kind of object as a literal. In this way if at runtime a message is to be sent to a node, this node can be pushed on the stack with a single `pushConstant` bytecode instruction. After that the send of `#markExecuted` can be directly performed. Benchmarks for the runtime penalty can be found further below.

Once we have done that we can run the code. Then each node is annotated with its execution count. To produce the final output we write a custom pretty printer that is a subclass of the standard pretty printer. If a node was executed it is printed green, otherwise it is printed red.

Listing 6.24: Instrumentation to mark nodes as executed

```
method instrument: [ :each |
    (self shouldInstrument: each) ifTrue: [
        each insertBefore: [ :node | node markExecuted ] ] ].
```

Benchmarks

In this section we investigate how much overhead per node code coverage causes. We benchmark the instrumentation with ByteNurse as well as the modified interpreter that visits the AST nodes. We use two different benchmark suites: the bytecode heavy benchmark in Listing 6.25 and the send heavy benchmark in Listing 6.26 that are part of the tinybenchmarks suite by Dan Ingalls.

Listing 6.25: Bytecode Heavy Benchmark

```
benchmark "Handy bytecode−heavy benchmark"
        "(500000 // time to run) = approx bytecodes per second"
        "5000000 // (Time millisecondsToRun: [10 benchmark]) * 1000"
        "3059000 on a Mac 8100/100"
    | size flags prime k count |
    size ←8190.
    1 to: self do:
        [:iter |
        count ←0.
        flags ←(Array new: size) atAllPut: true.
        1 to: size do:
            [:i | (flags at: i) ifTrue:
                [prime ←i+1.
                k ←i + prime.
                [k <= size] whileTrue:
                    [flags at: k put: false.
                    k ←k + prime].
                count ←count + 1]]].
    ↑count
```

- *Base* is the time for running the unmodified code in the VM.
- *Simulator* is the time for running the unmodified code in the interpreter simulator that simulates the execution of bytecodes.
- *Instrumentation without marking* is the time for running instrumented code that sends `#yourself` to each node before it is executed. This is

Listing 6.26: Send Heavy Benchmark

```
benchFib: anInteger
    "Handy send−heavy benchmark"
    "(result // seconds to run) = approx calls per second"
    " | r t |
      t ←Time millisecondsToRun: [r ←26 benchFib].
      (r ∗ 1000) // t"
    "138000 on a Mac 8100/100"
    ↑anInteger < 2
        ifTrue: [1]
        ifFalse: [(self benchFib: anInteger − 1) + (self benchFib: anInteger
            − 2) + 1]
```

used to determine the overhead of the instrumentation alone.

- *Marking using instrumentation* is the time for running the fully instrumented code that sends `#markExecuted` to each node before it is executed.
- *Interpretation without marking* is the time for running the code in the unmodified interpreter. This is used to determine the overhead of the interpretation alone.
- *Marking using interpreter* is the time for running the code with the modified interpreter that sends `#markExecuted` to each node before executing it.

Name	**Time**	**Factor**
Base	1 032	1
Simulator without marking	3 121	3
Instrumentation without marking	1 954	2
Interpretation without marking	9 321	9
Marking using instrumentation	3 204	3
Marking using interpreter	13 818	13

Table 6.4: Marking in the Bytecode Heavy Benchmark

In Table 6.4 we see that our interpreter is about three times slower than *Simulator*. We are satisfied with this performance considering how optimized the *Simulator* is and that it works on bytecodes. Compared to the VM the interpreter is only nine times slower. We think this is a consequence of the low level messages the benchmark sends that have a low send depth until they end up in a primitive so that the most time is spent there. Nevertheless instrumentation is considerably faster than interpretation and the resulting

code is only three times slower than the uninstrumented code. Additionally sending `#markExecuted` does not seem to increase the overhead much.

Name	Time	Factor
Base	1 620	1
Simulator without marking	120 424	74
Instrumentation without marking	3 967	2.4
Interpretation without marking	287 276	177
Marking using instrumentation	102 025	63
Marking using interpreter	502 686	310

Table 6.5: Marking in the Send Benchmark

Table 6.5 shows a much bigger difference between the Interpreter and the VM, almost factor 200. We explain this by the fact that the code spends less time in primitives and more time in actual Smalltalk methods. However the interpreter is only between two and three times slower than the Simulator. Again considering how optimized the simulator is we are satisfied with this performance. Instrumentation itself does not have a high penalty but sending `#markExecuted` dramatically increases the run time proving that the most time is spent there. Interpretation is considerably slower even without sending `#markExecuted` it is still slower than instrumentation with marking.

Concerning the performance of ByteNurse we conclude that "you only pay for what you use". Only nodes that are explicitly selected are instrumented. For all other nodes the generated code will not be changed. The cost of the setup, pushing the node on the stack and sending a very cheap message (`#yourself`), is only 140% in a very sensitive scenario and full instrumentation. Most of this time is actually used for doing the send. All the additional cost is caused by the added code itself. In the benchmark above with instrumentation and marking more than 96% of the time is spent in `#markExecuted`. Any improvement in the efficiency of that method directly translates into a better benchmark result.

Conclusion

ByteNurse allowed to easily build the instrumentations for code coverage by expression. The the performance of the resulting can be considered as fast enough for most cases. The major performance bottleneck was not the code generated by ByteNurse the `#markExecuted` method. The performance of the method `#markExecuted` could probably made considerably faster if keeping track of the execution count of a node was implemented with an instance variable instead of an annotation. This requires a conflicting change

to the node hierarchy but the speed gains might still justify it for certain applications. One major issue was to visualize the coverage results. A presentation engine for code as described in Section 4.1 whould have simplified this task considerably.

Although the performance of the interpreter is much lower than the VM it can still be considered as fast enough for many cases. Compared to the results of the simulator and considering how optimized the simulator is we are satisfied with its performance. This makes it well suited for building prototypes and hooking into the execution.

6.5 Pluggable Typesystem

TypePlug[5] is an optional, pluggable type system for Squeak. It consists of a type reconstructor and inferencer that is used by a type checker to check Squeak programs for type correctness.

Inside methods types are modeled as annotations on nodes in the AST. They can be declared on method and block arguments, method and block return values and temporary variable declarations. There are two different ways to declare types.

1. A special browser can be used. This has the advantage that type declarations can be made without changing the source code but still be checked into a source code management system. This is the preferred way for existing code especially system classes like `Boolean` or `Collection`.

2. Annotations can be placed in the source code. This is the preferred way for new code. It has the advantage that types can be written in the source code almost like in statically typed languages. The programmer can just normally type code and accept it. He is not required to use a special browser or do additional work after accepting a method.

For some statements like message sends to untyped code the type reconstructor is not able to inference a type. Because of this future versions of TypePlug will include the possibility to declare types on such statements, too. In the case of annotations in the source code no additional implementation effort is required. However it is a challenge for the browser that allows to do type declarations without changing the source code. This browser works with the standard low level representation of a method in the system that do not provide high level reflective facilities that reach into the method.

[5] `http://www.squeaksource.com/TypePlug.html`

Listing 6.27: TypePlug Example Code

```
aMethod: anArgument <:type: Boolean :>
    | aTemp <:type: Boolean :> |
    aTemp ←anArgument not.
    ↑([ :blockArgument <:type: Boolean :> |
        (blockArgument and: [ aTemp ]) not class ] value: aTemp)
        <:type: (Block args: {Boolean} return: Boolean class) :>
```

Listing 6.27 shows a method annotated with types. It takes a boolean as an argument. It has one temporary variable which is a boolean as well. It returns a block that takes one boolean as argument and returns a boolean class. Evaluation of annotation values in the source code at compile time can be used to construct the block type by sending `#args:return:` to `Block`.

A problem TypePlug faces is presentation of type mismatches. Ideally when the type reconstructor encounters a problematic node it could attach the error to it. A presentation plugin would later take care to underline the node and display the error message as a tooltip. This is not possible because we did not implement the presentation engine as described in Section 4.1. Currently the plan is to investigate if a modification of Shout[6] would provide the needed facility. This means returning to strings and a low abstraction level.

Future work includes doing instrumentation in order to enforce type correctness at runtime.

Conclusion

The feedback we received from the author of TypePlug was similar to the one of Section 6.3. He appreciated the high level of abstraction reflective methods provided and that they allowed him to focus on the problem domain instead of the implementation. He too reported that early versions suffered from stability problems that were corrected in later versions.

[6] `http://www.squeaksource.com/shout.html`

Chapter 7

Future Work

Several challenges remain for our implementation of reflective methods. One problem is the storage of the method including the AST. We see two possible options here. The first is writing flat files and the second is to use a database—either *relational databases* (RDBs) or *object-oriented databases* (OODBs). The advantage of text is that it is the traditional way of handling source code and thus supported by existing VCS. A drawback with text is that all annotations will have to be converted into text. It is important to note that this text is not what the programmer sees. It is more like HTML source code and the programmer sees and edits the rendered HTML. This makes it simpler to fix problems by hand, if something goes wrong. Databases have the advantage that they allow us to store objects directly without going back to text, a medium which we would like to avoid. Especially with RDBs the mapping can turn out to be a challenge because of the extreme polymorphism of the AST and the annotations. This makes us favor OODBs.

Once storage is implemented a VCS can be built that does not version files but semantic changes in the AST. If for example a selector of a message send has been changed, the VCS can capture this information. If an other programmer has put the same message send inside a block the VCS can combine those changes and merge them. Such information can be obtained in different ways. Most information can be computed automatically by comparing the differences between two trees. In cases where this fails the programmer himself can provide it, preferably with a tool. Another valuable source can be tools that manipulate code. For example if a variable is renamed with the refactoring engine it can attach this information to the node.

If special annotations that require specific classes are created then either the load order must be computed correctly or the construction of the annotations

must be delayed until the classes are loaded.

Furthermore special logic is required to preserve annotations when the programmer changes an existing method. We think this should be the responsibility of the tools at a higher level. For example, such a tool can implement breakpoints that are not `#halt` sends but annotations. The tool keeps track of all breakpoints and can deactivate them. It uses a plugin for the presentation engine to display the breakpoints to the programmer. An extended version allows conditional breakpoints where the annotation itself can contain code similar to an instrumentation annotation. This tool can also be used to recompile methods on certain events like key classes being changed as discussed in Section 6.1.

A presentation engine as described in Section 4.1 is needed in order to build a better, more configurable way to present source code based on the AST model. It needs to be extensible by plugins. There could be for example an SLint plugin that highlights expressions that violate a certain SLint rule. It offers a tooltip that shows the description of the violated rule and a menu with suggested corrections. Selecting one will cause it to be performed. Instrumentation related plugins can show code that was inserted or replaced in a special way. Other plugins could provide a mathematical rendering similar to TEXfor vectors and matrices. Also plugins that create non-textual views are conceivable.

Another major area is debugger and decompiler support. Right now the debugger tries to decompile the bytecode and then somehow locate the correct position in the source code. This is yet another place where we must go from one low level representation to an other. This does not yet work in all cases for the NewCompiler. In the presence of compiler plugins this will not work because the bytecode no longer matches the source code. But a debugger should be aware of instrumented code and present it as such to the programmer. It should offer special actions regarding such code like ignoring instrumentations and stepping into the original code or the opposite and stepping into the instrumented code. A prototype for such a debugger will probably use the interpreter that directly works on AST nodes. This way it already has a high level representation and must not start from bytecodes.

Geppetto [Röt06] currently uses ByteSurgeon as its back-end. It would be interesting to see if ByteNurse could be used and how it compares. Also is should be investigated if Geppetto can be used provide better MOP support like different implementation strategies for instance variable access, assignments or sends instead of the direct instrumentation at the expression level that ByteNurse currently does. Finally it would be interesting to see if Geppetto could profit from annotations for AST nodes or making the AST node available at the meta level.

One challenge is compiling the whole Squeak image with reflective methods. There are several issues to be considered here like memory consumption and access time. We built some prototypes using different approaches but never went beyond experimentation stage.

Although ByteNurse has matured much over time there is still room for improvement. Particularly when it comes to checking for valid input and and handling errors in a way that leaves the image in an unusable state. Also some additional metavariables like return value would be desirable. The integration of rewrite and search rules provided by the refactoring browser engine should be investigated. Automatically choosing the optimal compilation strategy for a method depending the present instrumentations would increase the ease of use.

Interesting future case studies include OODBs where instrumentation can be used to detect the modification of a persistent object and to redirect self sends in persistent objects or proxies. Zero runtime cost assertions and logging are also possible future case studies. Ideas for metaprogramming are the use of self modifying code for API migrations where a sender is refactored once it sends a deprecated message. Simple examples are method renames. A more complicated example is the migration from the old Seaside rendering API to the new one.

Although an AST has worked well for our experiments so far it might be that it is not well suited for some applications. We have not seen any other concrete and promising models, but should we find one we will explore it.

Chapter 8

Conclusion

We have implemented a causally connected model of methods that compared to the current model in Squeak provides a higher level of abstraction and has better support for structural reflection. These advantages combined with a convenient programming interface have enabled us to easily build tools that would have been very hard if not impossible with the current model. These tools have performed well in several case studies.

Annotations have worked well in the tools we implemented. They allow the tools to extend the model by adding data to it. In this way the tools can directly communicate by using the model. To add behavior, Smalltalk class extensions can be used. They have worked well for several years in the Smalltalk community. We feel the syntax for the source based annotations is a good compromise between easy parsebility, similarity to pragmas and minimal change to the Smalltalk syntax. It might look a bit strange first but a presentation engine should help to smooth things out. We are confident that annotations will scale well to bigger, more integrated case studies that include more tools and collaboration between them.

ByteNurse was used successfully in several real world case studies. The feedback we received from the authors was that it allowed them to easily build complicated instrumentations while having a very low runtime overhead. It allowed them to express instrumentations in an intuitive way. They could thus focus on their problem domain instead of the instrumentation. The reported stability problems on early version that were corrected over time.

A presentation engine for code based on an AST as described in Section 4.1 would have been invaluable. Every single case study presented in this thesis could have made good use of it.

It is interesting that the syntax of the programming language itself deter-

mines to a certain point the usefulness of our approach. If the syntax is extremely simple like LISP, then there will be only two types of node in the AST (a list node and an atom node). We believe this would make our model too general to be useful. If the syntax is extremely complex, there will be over 80 different node types in the AST. This automatically increases the complexity for the tool builder.

Based on the results of Chapter 6 we see our claims about ease of metaprogramming at the sub-method level and creation of tools confirmed and the restricted requirements fulfilled.

Appendix A

Installation

- Get a Squeak 3.9 image from `ftp://ftp.squeak.org/3.9/`
- Load the latest AST from `http://www.squeaksource.com/AST`
- Load the latest RefactoringEngine from `http://www.squeaksource.com/RefactoringEngine`
- Load the latest NewCompiler from `http://www.squeaksource.com/NewCompiler`
- Load the latest AST from `http://www.squeaksource.com/JCompiledMethods`
- Load the latest JCompiledMethods from `http://www.squeaksource.com/JCompiledMethods`
- enable the preference `#compileUseNewCompiler`
- optionally load other packages like Colorer, FastJavascript or JCompiledMethodsTest

Listings

List of Figures

List of Tables

Bibliography

[BD06] Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with Squeak. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[BJ00] Andrew P. Black and Mark P. Jones. Perspectives on software. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-oriented Systems*, 2000.

[BR] John Brant and Don Roberts. SmaCC, a Smalltalk Compiler-Compiler. http://www.refactory.com/Software/SmaCC/.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[Dek05] Steve Dekorte. Io: a small programming language. In Ralph Johnson and Richard P. Gabriel, editors, *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 166–167. ACM, 2005.

[Dim04] Sergey Dimitriev. Language oriented programming: The next programming paradigm. *onBoard Online Magazine*, 1(1), November 2004.

[Edw05] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In Ralph Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2004, San Diego, CA, USA*, pages 505–518. ACM, 2005.

[EK06] Andrew David Eisenberg and Gregor Kiczales. A simple edit-time metaobject protocol. In *International Conference on Aspect-Oriented Software Development*, 2006.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Han] Anthony Hannan. Squeak Closure Compiler. http://minnow.cc.gatech.edu/squeak/ClosureCompiler.

[HDD06] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

[LDGN06] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *CACM*, 3(4):184–195, April 1960.

[NDGL06] Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, and Adrian Lienhard. Analyzing, capturing and taming software change. In *Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[Qui03] Philip J. Quitslund. Beyond files: programming with multiple source views. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 6–9, 2003.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[Röt06] David Röthlisberger. Geppetto: Enhancing Smalltalk's reflective capabilities with unanticipated reflection. Master's thesis, University of Bern, January 2006.