# An Aspect Oriented Approach for Security Hardening: Semantic Foundations

Nadia Belblidia

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

December 2008

*Your file Votre référence*
*ISBN:* 978-0-494-63356-4
*Our file Notre référence*
*ISBN:* 978-0-494-63356-4

# ABSTRACT

## An Aspect Oriented Approach for Security Hardening: Semantic Foundations

Nadia Belblidia , Ph. D.

Concordia University, 2008

Computer security is nowadays a very important field in computer science and security hardening of applications becomes of paramount importance. Aspect oriented programming (AOP) is a relatively new technology that allows separation of concerns such as security, synchronization, logging, etc. This increases the readability, understandability, maintainability, and security of software systems. Furthermore, AOP allows automatic injection of the crosscutting concerns into the application code using a weaving mechanism. This thesis comes to provide theoretical study of using AOP for security hardening of applications. The main contributions of this thesis are the following. We propose a comparative study of AOP approaches from a security perspective. We establish a security appropriateness analysis of AspectJ and we propose new security constructs for this language. Since aspects in AspectJ are weaved (combined) with the Java Virtual Machine Language (JVML) application code, we develop a formal semantics for the JVML. We propose also a semantics for AspectJ that formalizes the advice weaving. We develop a new AOP calculus, λ_SAOP, based on lambda calculus extended with security pointcuts. Finally, we implement three new constructs in AspectJ, namely `getLocal`, `setLocal`, and `dflow`, for local variable accesses and data flow analysis. In conclusion, this thesis demonstrates the relevance, importance, and appropriateness of using the AOP programming paradigm in hardening the security of applications.

**Dedication**

To my parents who were my first and best teachers,

To my husband, who supported me patiently,

To my daughter who was always asking me if I had finished my Ph.D.. Now, I can tell her: "Yes, **I have finished**."

# ACKNOWLEDGEMENTS

The research for this thesis was carried out in the dynamic Computer Security Laboratory (CSL) of Concordia university. It has been a very instructive experience, professionally as well as personally. Now, the time has finally come to thank all the people that have contributed to this work. First, I would like to warmly thank my supervisor Pr. Mourad Debbabi. This work would not have been possible without his guidance, common-sense, knowledge, and perceptiveness. His suggestions and critical remarks, in combination with his strong vision and understanding, were very helpful. Thank you also Mourad for your continual moral support and encouragement. Many thanks also to the CIISE staff who provided for me an outstanding environment to work. I would like to say a big thank you to my colleagues: Dima Al-Hadidi, Mourad Azzam, Amine Boukhetouta, Aiman Hanna, Marc-André Laverdière, Syrine Tlili, and Zherong Yang. Thanks for the interesting scientific discussions about security analysis and hardening of free and open-source software. My appreciation also goes to all the friends that I made in the Institute along these years. Especially, I would like to thank Chamseddine Talhi for his help in proofreading my thesis to improve the written presentation of the final copy. Finally, and on a personal note, I wish to thank my parents, my sister, and my brothers for instilling in me confidence and a drive for returning to study for a Ph.D. after several years of work. Last but by no means least, I want to thank Fouzi, my husband, and Lylia, my only child for their patience and love during this long process. This work, and my life, would never have been the same without them.

# TABLE OF CONTENTS

# FIGURES

# TABLES

# ACRONYMS

| | |
|---|---|
| AJDT | AspectJ Development Team |
| AJC | AspectJ Compiler |
| AO | Aspect-Oriented |
| AOP | Aspect-Oriented Programming |
| AOSD | Aspect-Oriented Software Development |
| AP | Adaptive Programming |
| ASB | Aspect Sand Box |
| ASOC | Advanced Separation of Concerns |
| CSP | Communicating Sequential Processes |
| DRDC | Defence Research and Development Canada |
| EAOP | Event Aspect Oriented Programming |
| FTP | File Transfer Protocol |
| IEEE | Institute of Electrical and Electronics Engineers |
| JAAS | Java Authentication and Authorization Service |
| JCE | Java Cryptography Extension |
| JDT | Java Development Toolkit |
| JSAL | Java Security Aspect Library |
| J2EE | Java 2 Enterprise Edition |
| JVM | Java Virtual Machine |
| JVML | Java Virtual Machine Language |
| λ-Calculus | Lambda Calculus |
| λ-SAOP | Security Aspect Oriented Lambda Calculus |
| MAC | Message Authentication Code |
| MDSOC | Multi-Dimensional Separation of Concerns |
| OO | Object Oriented |
| OOP | Object Oriented Programming |

| | |
|---|---|
| SOS | Structural Operational Semantics |
| URL | Uniform Resource Locator |
| XSS | Cross-Site Scripting |

# Chapter 1

# Introduction

The aim of this thesis is to provide a formal and practical study of using AOP for software security hardening. This chapter presents the motivations of such a study, the research issues that we addressed, the objectives of the thesis, the methodology that we followed, the main contributions that we achieved, and finally the dissertation structure.

## 1.1 Motivations

Software security becomes increasingly important in this decade because of the growing connectivity of computers through the Internet. This growing interconnectedness of systems has increased the number of attacks and the facility with which an attack can be done. In addition, many applications deal with very sensitive data, such as corporate data, financial data, personal data, etc. It is then crucial to protect those applications because if hacked, they may cause significant damage.

Hardening the security of applications includes adding security functionalities as well as removing or preventing the exploitation of vulnerabilities. A legitimate question that one could ask is: "What is the most appropriate computation style or programming paradigm for security hardening?"A natural answer is to resort to an Aspect Oriented language. This answer is justified by the fact that the Aspect Oriented Programming

(AOP) [55] paradigm has been created to deal with the separation of concerns. Aspect Oriented Programming is a relatively new paradigm that complements the Object Oriented Programming (OOP) paradigm by supporting a better separation of crosscutting concerns[1]. Crosscutting concerns such as security are concerns that are tangled and scattered across more than one module.

The primary intent of this thesis is to elaborate a practical framework with the underlying solid semantic foundations for the application security hardening using the AOP paradigm. This work is motivated by several factors. The first is that application programmers can focus only on the functional problem they are facing. Using AOP, the base program does not need to consider security information, which can be put into separate and independent pieces of code. We can think that OOP also separates concerns by grouping them in separate objects; however, it is only appropriate to separate concerns that map to concrete objects not abstract concerns like security that affects the entire system in a broad manner. AOP allows security concerns to be specified modularly and applied to the main program.

Another motivation is that AOP is very interesting for security engineers because they would like to inject and strengthen security without digging into the logic of the application/middleware. This is made possible with the separation of concerns offered by the AOP approach. In addition, the injection of security modules is done in an automatic way.

This work is also motivated by the fact that correctness verification is much easier if the code is not scattered across many classes in the application.

For the semantic foundations of the approach, we motivate its usefulness with three facts. First, it is more suitable to use a formal semantics than an informal semantics to conduct fruitful discussions since it gives a better understanding of the concept. Second, a formal model allows to prove properties of the system. Finally, very few contributions related to the semantic foundations of AOP exist in the literature and our aim is to contribute

[1] AOP can be also on top of functional programming or even imperative programming

also in this field.

All these facts have motivated us to elaborate a research thesis targeting AOP-based application security hardening. The main challenge of this research is to assess the existing AOP approaches from a security point of view, choose one approach, extend it to better fit the problem of security hardening of applications, and to build the underlying semantic foundations.

## 1.2 Objectives

The purpose of this research is to elaborate a practical framework with the underlying solid semantic foundations for hardening the security of applications. More precisely, our objectives are:

- Study the practical and the theoretical underpinnings of existent aspect oriented techniques.
- Evaluate the appropriateness of the existent AOP approaches for security hardening and choose the most appropriate language for security hardening of Java applications.
- Design and implement a practical and efficient environment that allows the security hardening of Java applications.
- Define an AOP calculus for security hardening. Such a calculus will allow us to focus on AOP and security primitives without the details of a full-fledged language.

## 1.3 Research Issues

In this section, we present the research issues that we addressed at the practical and theoretical levels.

### 1.3.1 AOP for Security

When we started this research in the beginning of 2004, there were only few contributions in academic and industrial communities about using AOP for security hardening of applications. Most of these contributions use existing AOP languages to write security aspects or security libraries; however, they do not assess the AOP language they chose from a security perspective. Other contributions built safety-dedicated aspect extension of languages like C; however, these contributions do not define new and needed concepts in AOP and address local small-sized problems as buffer overflows and data logging. We found only two contributions [15,66] that analyze new relevant properties that can be used in AOP systems and identify areas of future work; However, none of them undertakes a fully-fledged language. This entailed the need to choose an appropriate AOP language for security hardening and to perform a complete security assessment of this language. In our case, the selected language is AspectJ.

### 1.3.2 AspectJ and Semantic Foundations

Establishing the semantic foundations for AspectJ entails the elaboration of a semantics for its advice weaving. Since the weaving in AspectJ is done on the JVML code, the first step is to define a semantics for JVML and then to define the semantics for the weaving. The most important issue we faced in the JVML semantics was to define a semantics for multi-threading. Notice that most of the proposed research contributions so far consider only one single thread of execution even though multi-threading is a keystone in Java. For this purpose, the semantics is structured in two layers: The first layer captures the semantics of sequential JVML programs in isolation, the second layer consists of judgements that capture the parallel execution of JVML threads.

Once the JVML semantics has been established, we had to develop a semantics for the weaving itself. For this purpose, we had to dig into the actual AspectJ implementation. The issue was that the source code of the AspectJ compiler is very huge and there is no

design document for it. Understanding the inner workings of the compiler was a hard task.

### 1.3.3 Security Aspect Calculus

When developing the aspect oriented calculus for security hardening, we used lambda-calculus. For this calculus, which we called λ_SAOP, we defined a semantics for the advice weaving during the typing process. We used the notion of effects and regions to control the type generalization in the presence of mutable data and tags to track data dependency in lambda expressions. This allows to find the join points that match the dataflow pointcuts. One of the challenges of this part of the work was to simultaneously use tags, effects and regions.

### 1.3.4 AspectJ Extension

A big challenge in this thesis is the extension of the Eclipse AspectJ compiler `ajc` [79] in order to handle the new proposed security primitives. In the literature, we did not find any documentation explaining the design of `ajc` compiler. Furthermore, the `ajc` code source, written in Java, contains more than 6000 Java files. It was challenging to find the most important files where we must intervene to design and implement the new primitives.

## 1.4 Methodology

To reach the objectives of the thesis, we followed this methodology:

### 1.4.1 Semantic Foundations of AOP

We spent time exploring the existent approaches to AOP, which encompasses the study of the existent mainstream languages: Aspect J, Hyper J, and DJ. To achieve a great understanding of these languages, our sources were the available informal specifications

as well as the existing compilers. In addition, existing research papers on these topics have been studied in order to gain more understanding and insights into these approaches. As for the theoretical aspects underlying AOP, we studied the very few published works on the semantic foundations.

To benefit from the effort invested in studying the state of the art of AOP, we dedicated research efforts to compile the know-how of the AOP community into an elegant semantic framework.

Actually, we noticed that a tremendous work has been done in the related work on designing and implementing very sophisticated AOP languages. These languages include very expressive and powerful primitives while their semantics are very complex to grasp. To fully understand the meaning of these primitives we had to:

- Dig into the actual implementations of the corresponding programming languages. This task has been very tedious and time consuming since it implies reading thousands of lines of code without necessarily having the design documentation.

- Scrutinize both the source code of programs and the corresponding compiled units in order to figure out how these primitives have been interpreted by the compiler. This implies the design of relevant testing programs and the decompilation/disassembling of the compiled units that are the result of the compilation/weaving process. This has been again a very tedious and time consuming task.

Once understood, we compiled this tremendous knowledge into an elegant, formal, rigorous and robust semantic framework.

### 1.4.2 Security Appropriateness of AOP

An important objective was to assess the adequacy of existing AOP approaches to handle security hardening. To achieve this, we had to:

- Compile and analyze common best practices in security hardening.

- Determine if these practices are expressible in terms of the existing AOP primitives.

As we will explain later in this thesis, existing approaches are not expressive enough to capture security hardening practices; therefore, the aforementioned research task will lead to the identification of relevant and new AOP security primitives.

### 1.4.3 Language for Security Hardening

In this phase of our research, we took AspectJ with its compile-time and run-time definitions and extended it with the main important AOP primitives identified and elaborated in the previous phases of the research. This entailed the modifications of both the compiler (more accurately the parser, semantic analyzer, and code generator) and the weaver. Notice that these modifications are the result of a joint collaboration with other CSL (Computer Security Laboratory) members as part of a larger project funded by Defense Research and Development Canada (DRDC) and Bell Canada under NSERC partnership.

### 1.4.4 AOP Calculus for Security

To fully understand the semantics of AOP security hardening and in order to establish theoretical results on this theme, we elaborated the syntactic and semantic definitions of a calculus for security hardening called λ_SAOP Calculus. Recall that calculi are compact, expressive, and algebraic languages that are elaborated for the main purpose of understanding a computation style and to establish theoretical results.

To elaborate the calculus, we took advantage of the previously described research results, i.e.:

- The theoretical foundations of AOP languages.
- The identification of AOP security hardening primitives.

The syntactic definition encompasses (1) Primitives for the functional and computational fragments of the calculus and (2) AOP Primitives, i.e., pointcut and advice constructs. We used the semantic definitions in order to establish an important result, which ensures that the advice weaving does not change the types, i.e., the original program and the weaved program have the same type. This is an important correctness property that the weaving process should satisfy.

## 1.5 Contributions

This section describes the different contributions achieved during this thesis. We have elaborated:

- A security appropriateness analysis of the three important AOP approaches: pointcut-advice model [56], adaptive programming [76], and multiple separation of concerns [78]. We concluded [2] that the pointcut-advice model was the best for security hardening. The most popular AOP language that extends Java and based on the pointcut-advice model is AspectJ. We studied in detail AspectJ and its security appropriateness. We proposed [1] extensions to AspectJ in order to successfully address the programming of security concerns.

- A dynamic semantics for the Java Virtual Machine Language because the weaving in AspectJ is done on the JVML code. The presented semantics [11] is a faithful and formal transcription of the JVML specification as described in [64]. The semantics is structured in two layers: The first layer is devoted to the semantics of sequential JVML programs and the second layer captures the parallel execution of JVML threads.

- A formal semantics of the AspectJ advice weaving [9, 10], which compiles the know-how of the AspectJ community. This semantics covers both static and dynamic pointcuts. A static pointcut is a pointcut dealing only with compile time

information whereas a dynamic pointcut needs runtime information. In order to build this semantics, we had to scrutinize both the source code of programs and the corresponding compiled units in order to determine how the AspectJ primitives are interpreted by the compiler.

- A security aspect calculus λ_SAOP [3, 4] containing primitives for the functional and computational fragment of the calculus and AOP primitives. The calculus contains pointcuts that are relevant to security hardening of applications. We develop a semantics for this calculus and we propose an accommodation to the effect-based inference algorithm to take the matching and the weaving processes into consideration.

- An extension of the Eclipse AspectJ compiler `ajc` version 1.5. This extension [5] implements new pointcuts that are relevant from security point of view. The considered pointcuts are related to local variable accesses, namely `getLocal` and `setLocal`, and to data flow information analysis, namely `dflow`.

## 1.6 Thesis Structure

The rest of this document is organized as follows. Chapter 2 gives an overview of the background and the related work of the topics discussed in this thesis. Chapter 3 presents a security appropriateness analysis of the different AOP approaches and more accurately of AspectJ. Chapter 4 presents a semantics for JVML. Chapter 5 provides a semantics for the advice weaving in AspectJ. Chapter 6 presents the security functional AOP calculus λ-SAOP. Chapter 7 details the implementation of the primitives `getLocal`, `setLocal`, and `dflow` in AspectJ. Finally, in Chapter 8, we give some conclusions and future work on this research .

# Chapter 2

# Aspect Oriented Programming and Security

The intent of this chapter is twofold, firstly to present the background information related to the topics discussed in this thesis and secondly, to give the most relevant contributions to the subject of the thesis. Concerning the background, we will first outline the basic principles of software security. Next, we will familiarize the reader with the aspect oriented approaches. Thirdly, we will describe AspectJ specificities. Finally, we will provide an overview of lambda calculi and type systems. We remind the reader that the security AOP calculus λ_SAOP that we present in Chapter 6 is based on lambda-calculus. Concerning the related work, first we will cite the contributions where AOP is used for security goals then the contributions that formalize the semantics of AOP languages followed by those that describe semantics for JVML.

## 2.1 Software Security

A software is secure if it resists to unintentional defects, accidents, and failures as well as intentional attacks. Software security is the ability of software to resist to these kind of events. Attacks can be initiated either inside or outside the site of an organization and are

violation of one or more security properties. In this section, we review the background about high-level and low-level security properties as well as the security hardening practices.

### 2.1.1 High-Level Security

While designing and developing applications, we should take care of different high-level security properties. Several security properties related to information security are described in many documents, standards, and books but there is no standard documentation listing all these security requirements. Each security property has its own characteristics. Among the different properties, we can enumerate: confidentiality, integrity, availability, authentication, authorization, non-repudiation, anonymity, and accountability. A presentation of these security properties is given hereafter:

1. *Confidentiality*: The confidentiality property specifies that critical information should not be made available or disclosed to unauthorized individuals, entities, or processes. The system enforcing confidentiality shall offer features to ensure that only authorized users are allowed access sensitive information.

2. *Integrity*: The integrity property specifies that information should not be modified or destroyed in an unauthorized manner.

3. *Availability*: The availability property specifies that unauthorized persons or systems cannot deny access/use by authorized users.

4. *Authentication*: The authentication property corroborates that an entity is who it claims to be. To ensure authentication, the system must offer a way to verify the identity of a user before allowing her or him access to the system.

5. *Authorization*: The authorization property (known as access control) is concerned with restricting access to resources to privileged entities. To enforce authorization,

the system shall offer features to not allow a user to access a resource of the system unless authorized to do so.

6. *Non-repudiation*: Enforcing the non-repudiation property prevents the denial of participation in transactions. To enforce non-repudiation, a system must have the capability of preventing users from denying actions and events of other users acting as senders or receivers.

7. *Anonymity*: The anonymity property specifies that the identity of the user participating in a certain event is kept secret. This property may be needed in many situations, like voting, posting messages on forums, etc. Notice that anonymity is different from confidentiality since confidentiality keeps the content of a message secret whereas anonymity keeps secret the identity of the originator or that of the recipient. To ensure anonymity, a system must have the ability to hide identities.

8. *Accountability*: The accountability property specifies that all the actions of a system entity (an invoker of a service, an initiator of action) may be traced. To enforce accountability, a system must have tools to authenticate the users.

Other security objectives exist in the security literature such as auditing, certification, and so forth. Several security mechanisms are used to enforce the aforementioned requirements, such as cryptographic protocols (based on symmetric or asymmetric keys) for confidentiality and integrity, MAC (Message Authentication Code) and digital signatures for authentication and so forth.

High-level security is very important; however, it is not always sufficient. For instance, problems as buffer overflows are not covered by high-level security properties. The following section discusses such problems in detail.

### 2.1.2 Low-Level Security

The role of the low-level security (safety) mechanisms is to ensure that a component is "safe to use". Low-level security is extremely dependent on the programming language and the platform. In the case of Java, safety issues are minimal because this language provides concepts such as strong typing, no address arithmetics, array bound-checking, and exception handling that makes it safe. Most Java related security problems are about high-level security properties. In the case of the C and C++ languages, the lack of type safety and the fact that memory management is left to the programmer's discretion are the major causes of low-level security vulnerabilities. In the sequel, we introduce the most important safety vulnerabilities that can be introduced in the source code during the implementation.

1. *Buffer Overflow*: Buffer overflows are the best known software security vulnerabilities and the tool of choice of attackers. A buffer overflow occurs when data is written outside of the boundaries of the memory allocated to a particular data structure. Buffer overflow vulnerabilities are generally used to overwrite stack pointers and to change the program flow in order to execute malicious instructions. The impact of buffer overflow vulnerabilities is big since attacks exploiting these vulnerabilities can compromise the integrity, confidentiality, availability, and other security factors of the targeted system.

2. *Integer Overflow*: Integer overflow is an other kind of memory corruption vulnerabilities. Integer overflow issues occur when an application tries to create a numeric value that is too large for the available storage space. Integer overflow is often undetected by applications, which may lead to a security breach through a buffer overflow or other malicious code.

3. *Format String*: Format string vulnerabilities are also exploited for memory corruption. A format string is a string that describes how a specific output should be formatted. Many languages allow format string but it is a vulnerability in C. Indeed,

printing functions in C do not check the number or types of their variable arguments. Hence, an attacker that inserts unexpected `%n` symbols into user-supplied input strings can perform unauthorized writes into the memory. He may use also `%s` and `%x` formats to read data from the memory. Format string attacks can be used to crash a program or to execute harmful code.

4. *Temporary file races*: The temporary file race vulnerability is the most known TOCTTOU (Time of Check to Time of Use) binding flaws. The idea is that a privileged program first probes the state of a file (time to check), and then based on the results of that check, takes some actions (time to use). This is is vulnerable to a race because the two actions are not atomic. Hence, an attacker may exploit this vulnerability by "racing" between the probe and the action to change by making for example a link to a file he wanted to modify.

5. *Memory Management*:

   Memory management vulnerabilities are due to mistakes in coding memory management. An inadequate use of the dynamic memory management functions such as `malloc`, `calloc`, and `free` in standard C can lead to vulnerabilities resulting in writing to freed memory, buffer overflows, and freeing the same memory multiple times. Thus, memory management functions must be used with precaution in order to prevent unauthorized access to memory space, memory corruption, etc.

### 2.1.3 Security Hardening Practices

In information technology, security hardening is the process of protecting a system against threats. It can be described as adjusting system configuration and software in order to reinforce the software security. Software security hardening consists in adding security functionalities, removing vulnerabilities and preventing their exploitation in the software. In spite of its importance, software security hardening has not seen as much research or effort as other areas of information security technologies like firewalls, intrusion detection,

etc. We list in the following the most important application security hardening methods:

1. *Code-Level Hardening*: Code-level hardening consists of changing the source code in order to prevent vulnerabilities and/or to add high-level security properties. More precisely the code improvements apply proper coding standards that were not applied initially.

2. *Software Process Hardening*: Software build process is the process of translating source code into binary code, linking the different modules, creating libraries, and assembling the binary modules into programs. Hence, software process hardening is the addition of security features via the software build process without changing the original source code. This is performed using compiler options and including more secure versions of library linking.

3. *Design-Level Hardening*: Design-Level hardening consists of re-engineering the underlying application architecture in order to enhance the application security. The application design and specification are changed in order to improve existent security features or to introduce new ones. Indeed, some flaws are due to weaknesses in the design and a re-engineering of the application is the only way to fix them (flaws). Notice that this category of hardening practices is more about high-level security.

4. *Operating Environment Hardening*: Operating environment hardening consists in improving the execution context of the application. It is not related directly to the software but has an impact on it since it addresses its execution environment. For example, operating system hardening (via configuration) provides a better protection against corruption and prevents from bypassing the application security procedures. Other examples are the protection of the network layer, the use of high-security dynamically-linked libraries, the use of security-related operating system extensions, etc.

In the scope of this research, we are interested by the code level hardening of Java applications. As mentioned before, Java security is mostly about high-level security.

## 2.2 Aspect Oriented Programming

Separation of concerns is a general principle in software engineering, introduced by Dijkstra [34] and Parnas [81]. It enables us to break the complexity of a problem into loosely-coupled subproblems that are easier to solve and afterwards solve the subproblems in isolation and combine the resulting modules into one solution. This has several advantages: reducing software complexity, improving their comprehensibility, promoting traceability and facilitating reuse and evolution. Thanks to object oriented programming (OOP), developers can now produce more modular implementations of complex systems. However, it is still sometimes difficult or impossible to achieve a good separation of concerns using only OOP.

Aspect Oriented Programming (AOP) [55] is a new programming paradigm that allows modular implementation of crosscutting concerns. A crosscutting concern is one which is tangled with others in a way that cannot be easily separated using only OOP techniques. A typical example of crosscutting concern is the "logging" example that helps debugging or other purposes by tracing method calls. Assuming that we do logging at the beginning and the end of each function call, this will result in crosscutting all the classes where functions are invoked. The AOP approach allows to separate the concern of "logging" and to structure it into a single unit called an aspect.

Many AOP languages have been developed and the most prominent are AspectJ [56] and HyperJ [91], which are built on top of the Java programming language. There has also been work done to provide AOP frameworks for other languages. AspectC [24] is an extension to C that is used to provide separation of concerns in operating systems. Similarly, AspectC++ [86] and AspectC# [57] are AOP extensions of the C++ and C# languages, respectively. For the Smalltalk language, a version of AOP has been presented

in [16]. Those languages adopt various approaches of AOP and present significant differences due to the abstraction mechanisms, modules and language extension, and specific language constructs they use.

All the AOP approaches are based on the notion of weaving. Weaving is the process of composing core functionality modules with aspects into one single application. In the following section, we will present the three most known approaches.

### 2.2.1 Pointcut-Advice Approach

The pointcut-advice approach is based on the mechanisms of pointcuts and advices. A pointcut is a set of possible places called join points at which an aspect may specify its behavior whereas the advices represent the corresponding behavior. AspectJ [56], an aspect oriented extension of Java, is probably the most known representative of pointcut languages and AOP languages in general. Developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center), AspectJ is currently a part of an openly developed Eclipse project.

Figure 2.1 depicts a very simple example written with AspectJ. It contains two Java classes: `MyClass` and `Tester` and an aspect `Logger`. In this aspect, the pointcut `callSayMessage` represents the set of all join points that correspond to any method execution. The `before` and `after` advices print to the standard output the identification of the join point before and after the execution of the method. The entire execution of the example is as follows:

```
>>Enter to execution (void Tester.main(String[]))
MyClass's Constructor
>>Enter to execution (void MyClass.m1())
Good Morning from m1
>>Out from execution (void MyClass.m1())
>>Out from execution (void Tester.main(String[]))
```

```
public class MyClass {
public MyClass(){
  System.out.println("MyClass's Constructor");
}
public void m1() {
  System.out.println("Good Morning from m1")
}
}
public class Tester {
public static void main(String[] args) {
  MyClass t = new MyClass();
  t.m1();
}
}
public aspect Logger {
pointcut callSayMessage(): execution (* *.*(..));

before(): callSayMessage() {
System.out.println(">>Enter to " + thisJoinPoint);
}
after(): callSayMessage() {
System.out.println(">>Out from " + thisJoinPoint);}
}
}
```

Figure 2.1: AspectJ Example1

AspectC, AspectC++ and AspectC# are also based on the pointcut-advice model. To have a better understanding of this model and more precisely of AspectJ, we refer the reader to Section 2.3 where AspectJ is presented in more details.

### 2.2.2 Multi-Dimensional Separation of Concerns

Multi-dimensional separation of concerns (MDSOC) allows simultaneous separation according to multiple, arbitrary kinds (dimensions) of concerns [78]. MDSOC treats all concerns as first-class and co-equal, including components and aspects, allowing them to be encapsulated and composed. This is in contrast to most aspect oriented approaches, which enable aspects to be composed from components but do not support composition

of components (or aspects) with one another [92].

One particular approach of MDSOC is called hyperspaces where the software is modelled as a set of units called hyperslices (concerns). A hyperslice is a set of modules representing only one single concern and could be composed with other hyperslices using matching units (e.g. method names).

HyperJ [78, 91] is the support for hyperspaces in Java. Developed at the IBM Thomas J. Watson Research Center, HyperJ allows developers to decompose and organize code according to several criteria simultaneously even after the implementation of the software. HyperJ operates on and generates standard Java class files and uses separate configuration files for the weaving instructions. When using HyperJ, the developer has to specify:

- The Java programs.
- A hyperspace file that will enumerate the Java files to consider and that will be manipulated by HyperJ.
- A concern mapping that identifies for each dimension of concerns the corresponding pieces of code (packages, classes and/or methods).
- An hypermodule file that describes which dimensions of concerns (hyperslices) are involved and that specifies the composition relationships.

We illustrate the use of HyperJ with the Java programs in Figure 2.2 and Figure 2.3.

The constructor and the method `m1` of the class `MyClass` in the package `French`, respectively in the package `English`, simply print specific string in French, respectively in English, to the standard output. For a first example, we use the hyperspace, the concern mapping and the hypermodule described in respectively Figure 2.4, Figure 2.5 and Figure 2.6.

The execution of this first example is as follows:

```
Constructeur de MyClass
MyClass Constructor
```

```
package French;
public class MyClass {
public MyClass(){
  System.out.println("Constructeur de MyClass");
}

public void m1() {
  System.out.println("Bonjour de m1");
}

}

public class Tester {
public static void main(String[] args ) {
  MyClass t = new MyClass();
  t.m1();
}

}

public class Logger {
public void invokeBefore() {
  System.out.println("Avant appel de m");
}
public void invokeAfter() {
  System.out.println("éAprs appel de m1");
}

}
```

Figure 2.2: HyperJ Example: Java Programs Part I

```
package English;
public class MyClass {
public MyClass() {
  System.out.println("My Class Constructor");
}

public void m1() {
  System.out.println("Good Morning from m1");
}

}
```

Figure 2.3: HyperJ Example: Java Programs Part II

```
hyperspace Hs1
  composable class French.*;
  composable class English.*;
```

Figure 2.4: HyperJ Example: HyperSpaceFile

```
package French: Feature FrenchStuff
package English: Feature EnglishStuff
```

Figure 2.5: HyperJ Example: Concern Mapping

```
hypermodule Sample1
  hyperslices:
    Feature.FrenchStuff,
    Feature.EnglishStuff;
  relationships: mergeByName;
end hypermodule;
```

Figure 2.6: HyperJ Example: HyperModule 1

```
Bonjour de m1
Good Morning from m1
```

The relationship `mergeByName` indicates that units in different hyperslices that have the same name are to be integrated together into a new unit. The integration is done in the same order than the appearance of the hyperslices in the hypermodule file. HyperJ

allows many other strategies as `overrideByName`, which indicates that units with the same name are to correspond, and are to be connected by an override relationship, which causes the last one to override the others in the composed software.

In a second example, we use the same Java programs and configuration files except that we will use the hypermodule of Figure 2.7 where we specify an order on the methods instead of Figure 2.6.

```
hypermodule Sample1
  hyperslices:
    Feature.FrenchStuff,
    Feature.EnglishStuff;
  relationships: mergeByName;
  order action Feature.EnglishStuff.MyClass.m1 before
              action Feature.FrenchStuff.MyClass.m1;
end hypermodule;
```

Figure 2.7: HyperJ Example: HyperModule 2

The execution presented hereafter shows how the weaving is done in this case and how the order of the merge for the methods `m1` is different from the first example where no order was specified.

```
Constructeur de MyClass
MyClass Constructor
Good Morning from m1
Bonjour de m1
```

In a last example, we use the hypermodule of Figure 2.8. The bracket relationship indicates that a set of methods should be preceded and/or followed by other specified methods. Indeed, in the bracket relationship of the example, the method `m1` of class `MyClass` will be bracketed between the methods `invokeBefore` and `invokeAfter`.

In this case, the result of the weaving is as follows:

```
Constructeur de MyClass
MyClass Constructor
Before the Call of m1
```

```
hypermodule Sample1
  hyperslices:
    Feature.FrenchStuff,
    Feature.EnglishStuff;
  relationships: mergeByName;
  bracket "MyClass"."m1" before Feature.FrenchStuff.Logger.
     invokeBefore( $OperationName, $ClassName),
    after Feature.FrenchStuff.Logger.invokeAfter($OperationName,
       $ClassName);
end hypermodule;
```

Figure 2.8: HyperJ Example: HyperModule 3

```
Bonjour de m1
Good Morning from m1
After the Call of m1
```

For further details on the other strategies of HyperJ, the reader is referred to its reference manual [91].

### 2.2.3 Adaptive Programming

Adaptive programming(AP), proposed by Demeter group [28] at Northeastern University in Boston, has used the ideas of AOP several years before the name Aspect Oriented Programming was coined. The law of Demeter is a programming style rule for loose coupling between the structure and behavior concerns. The Demeter law states that any method *M* of a class *C* should use only the immediate part of the object, the parameters of *M*, objects that are directly created in *M* or global objects. Any program can be modified to conform with the Demeter law, but the drawback of following this law is that it can result in a large number of small methods scattered throughout the program. This can make it hard to understand the program's high-level picture. Adaptive programming with traversal strategies and adaptive visitors avoids this problem while better supporting the loose coupling of concerns [76]. The Demeter method has three steps [75]:

- Derive a class graph: Starting from the requirements create a set of classes that best

captures the structure of the program data

- Derive traversal methods: For each use (program operation), find a traversal path by specifying the root of the traversal, the target classes, and the constraints in between to restrict the traversal.

- Derive visitor methods: Attach specific behavior to certain classes that are visited along each traversal. This is the "meat" of your program.

DJ [77] is a free implementation of adaptive programming for Java that supports this style of programming. DJ allows to traverse an object graph according to the traversal strategy and to specify a visitor to be executed before and after each traversed node. In the example depicted in Figure 2.9, the traversal graph corresponds to the subgraph starting from the node `A` to the node `C` via the node `B`. The methods `before` and `after` are executed before and after the traversal of a matching object. The execution of this example is:

```
Entering A
Entering B
Entering C
Exiting C
Exiting B
Exiting A
```

## 2.3 AspectJ

This section is devoted to a detailed description of AspectJ. AspectJ [56] is an AOP language that has been released in 1998. It emerged from a research work at Xerox PARC on the aspect oriented programming paradigm in the 80s and 90s. Now, AspectJ is developed as part of the Eclipse Project. AspectJ has been designed with the objective of

```
import edu.neu.ccs.demeter.dj.*;
class Main {
public static void main(String[] args) {
  ClassGraph cg = new ClassGraph();
  // constructed by reflection from the classes in default
     package
  A a = new A(new B(new D(),new C()), new C());
  Strategy sg= new Strategy("from A via B to C ");
  TraversalGraph tg = new TraversalGraph(sg,cg);
  // Visitor is used in the following
  tg.traverse(a,new Visitor(){
  public void start() { System.out.println("begin"); }
  public void finish() { System.out.println("end"); }
  public void before(A o) { System.out.println("Entering A");}
  public void after(A o) { System.out.println("Exiting A"); }
  public void before(C o) { System.out.println("Entering C");}
  public void after(C o) { System.out.println("Exiting C"); }
  public void before(D o) { System.out.println("Entering D");}
  public void after(D o) { System.out.println("Exiting D"); }
  public void before(B o) { System.out.println("Entering B");}
  public void after(B o) { System.out.println("Exiting B"); }
  });
}
}
```

Figure 2.9: DJ Example

being an easy-to-learn and easy-to-use language. It is a conservative extension to Java: Every valid Java program is a valid AspectJ program. Furthermore, AspectJ compiles to normal Java bytecode that can be executed in a standard JVM, not requiring a specialized runtime environment. AspectJ supports two kinds of crosscutting implementation: the dynamic crosscutting and the static crosscutting. The first allows us to define additional behavior to run at join points. The second affects the static type signature of the program and allows to define new operations on existing types. These crosscutting behaviors are encapsulated in an AspectJ construct known as an aspect.

### 2.3.1 Dynamic Crosscutting

This section describes AspectJ's dynamic crosscutting, which is based on a powerful set of constructs: Join points, pointcuts, advices, and aspects. Join points are well-defined points in the execution of the program. Pointcuts describes collections of join points. Advices define the additional behavior at the join points and are method-like constructs. Aspects are units composed of pointcuts, advices, and ordinary Java member declarations. Similar to a Java class, an aspect can contain both fields and methods but it cannot be explicitly instantiated.

We detail in the following the join point model, the different pointcut designators, and the different kinds of advices in AspectJ.

#### Join Point Model

AspectJ provides many kinds of join points. Join points are the points where the integration of crosscutting concerns is done. Among the different join points that a program may have, AspectJ exposes and defines only the join points described in Table 2.1.

AspectJ provides a special variable called `thisJoinPoint`, which contains the dynamic information associated with join points. This variable allows us to get information on the target object, the executing object, and the method arguments. We can also get other information using the variable `thisJoinPoint` such as the name of the executing method using the Java reflection API.

#### Pointcut Designators

A pointcut is a construct that picks out join points and exposes data from the execution context of those join points. In a simple way, we can think of pointcut designators in terms of matching certain join points at runtime. For instance, the pointcut designator `call(void Point.f(int))` matches all method call join points where the Java signature of the called method is `void Point.f(int)`. AspectJ supports both named

| *Joinpoint* | *Meaning* |
|---|---|
| Method execution | When the body of code for an actual method executes. |
| Method call | When a method is called, not including super calls of non-static methods. |
| Constructor execution | When the body of code for an actual constructor executes, after its this or super constructor call. |
| Constructor call | When an object is built and that object's initial constructor is called. |
| Field reference | When a non-constant field is referenced. |
| Field set | When a field is assigned to. |
| Advice execution | When the body of code for a piece of advice executes. |
| Object initialization | When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. |
| Static initializer execution | When the static initializer for a class executes. |
| Object pre-initialization | Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. |
| Exception-handler | Start is found from exception handler table. |

Table 2.1: AspectJ Join Points

and anonymous pointcuts. Named pointcuts are declared with the keyword `pointcut` and can be used in several places in the aspect.

Pointcuts can be composed with the operators `&&`, `||`, and `!` to build other pointcuts using respectively conjunction, disjunction, and negation of pointcuts. The primitive pointcuts provided by the language are represented in Tables 2.2 and 2.3 following the documentation on the Eclipse site [80].

An AspectJ pointcut is either a static pointcut or a dynamic pointcut. A static pointcut describes join points that can be determined by a static analysis whereas a dynamic pointcut describes join points that cannot be determined statically. The following AspectJ pointcut is a static one and describes all the join points that are in the class `A` and call the void method `logging` of class `B`:

```
pointcut callLoggingFromAtoB():
        call(void B.logging()) && within(A);
```

whereas the next pointcut depends on the type of the executing object. A call of a void method `logging` in a superclass of `A` might be a valid join point if the object is an instance of `A`.

```
pointcut callLoggingFromA():
        call(void *.logging()) && this(A);
```

A static pointcut can be directly mapped to code and the matching process knows at compile time if a join point matches a given pointcut or not. For dynamic pointcuts, the weaver inserts, when necessary, JVML test instructions to check join point runtime properties. AspectJ supports three kinds of dynamic tests: tests based on the execution flow, tests evaluating a boolean expression and the `instanceof` tests that check object types at runtime.

### Advices

Advices are used to implement crosscutting behaviors. Indeed, the pointcuts alone can only pick out join points and do not add any behavior to the base application. Each

| *Pointcut* | *Meaning* |
|---|---|
| `call`(*MethodPattern*) | Picks out each method call join point whose signature matches *MethodPattern*. |
| `execution`(*MethodPattern*) | Picks out each method execution join point whose signature matches *MethodPattern*. |
| `get`(*FieldPattern*) | Picks out each field reference join point whose signature matches *FieldPattern*. |
| `set` (*FieldPattern*) | Picks out each field set join point whose signature matches *FieldPattern*. |
| `call`(*ConstructorPattern*) | Picks out each constructor call join point whose signature matches *ConstructorPattern*. |
| `execution`(*ConstructorPattern*) | Picks out each constructor execution join point whose signature matches *ConstructorPattern*. |
| `initialization`(*ConstructorPattern*) | Picks out each object initialization join point whose signature matches *ConstructorPattern*. |
| `preinitialization`(*ConstructorPattern*) | Picks out each object pre-initialization join point whose signature matches *ConstructorPattern*. |
| `staticinitialization`(*TypePattern*) | Picks out each static initializer execution join point whose signature matches *TypePattern*. |
| `handler`(*TypePattern*) | Picks out each exception handler join point whose signature matches *TypePattern*. |
| `adviceexecution`() | Picks out all advice execution join points. |
| `within`(*TypePattern*) | Picks out each join point where the executing code is defined in a type matched by *TypePattern*. |
| `withincode`(*MethodPattern*) | Picks out each join point where the executing code is defined in a method whose signature matches *MethodPattern*. |

Table 2.2: AspectJ Pointcuts Part I

| *Pointcut* | *Meaning* |
|---|---|
| `withincode`(*ConstructorPattern*) | Picks out each join point where the executing code is defined in a constructor whose signature matches *ConstructorPattern*. |
| `cflow`(*Pointcut*) | Picks out each join point in the control flow of any join point *j* picked out by *Pointcut*, including *j* itself. |
| `cflowbelow`(Pointcut) | Picks out each join point in the control flow of any join point *j* picked out by Pointcut, but not *j* itself. |
| `this`(*Type* or *Id*) | Picks out each join point where the currently executing object is an instance of *Type*, or of the type of the identifier *Id*. |
| `target`(*Type* or *Id*) | Picks out each join point where the target object is an instance of *Type*, or of the type of the identifier *Id*. |
| `args`(*Type* or *Id*, ...) | Picks out each join point where the arguments are instances of the appropriate type. |
| `if`(*BooleanExpression*) | Picks out each join point where the boolean expression evaluates to true. |

Table 2.3: AspectJ Pointcuts Part II

advice brings together a pointcut and a body of code to run when join points matching the pointcut are reached. Hence, an advice is a method-like mechanism that is used to declare certain code that should execute at each of the join points matching its pointcut. AspectJ supports three kinds of advice before, after, and around advice. A before advice runs when the join point is reached but before the program proceeds with the join point. After advice runs after the program proceeds with the join point. For example, after advice on a method call join point runs after the method body has run. Notice that there are two special cases of after advice: after returning and after throwing. In fact, Java programs can leave a join point either normally or by throwing an exception. The plain `after` advice runs after returning or throwing. The `before` advices and `after` advices have only additive capacity whereas around advice can preempt the normal computation of a join point and has explicit control over whether the program proceeds with the join point or not. At each join point, the advices are examined in order to see whether they apply to the join point.

Several advices may apply to a same join point and the term used in the AOP community for such cases is "Aspect Interference". In AspectJ, the programmer can explicitly define a precedence order between aspects. The precedence relationship is then declared using the following syntax:

`declare precedence:` *TypePatternList* ;

Hence, if two advices from two aspects in *TypePatternList* are to be applied on the same join point, then the precedence of the advice will follow the order in the list. For example, if we want that aspects with security as part of their names have precedence on all the other aspects, we will express this by:

```
declare precedence:  *..*Security*, *;
```

Since AspectJ 1.1, the advice weaving has been based on bytecode transformation rather than on source code transformation. It is done statically by inserting the advice functionality in certain regions, called join point shadows, in the program that correspond to the join points matched by the advice pointcut. In this case, the AspectJ compiler is composed of two stages. The first stage (front-end compiler) is implemented as an extension to the Java compiler and compiles applications and aspects into pure Java bytecode enriched with additional annotations to handle non pure Java information as advices and pointcuts. The second stage (back-end compiler) weaves compiled aspects with compiled applications producing woven class files.

### 2.3.2 Static Crosscutting

Advice declarations, as seen in the previous section, do not allow changing static type structure of a class. Indeed, they only change the behavior of classes they crosscut. In order to allow changing classes static structure, AspectJ offers other tools as inter-type member declarations and other declare forms. Those forms allow:

- Adding methods to an existing class.
- Adding fields to an existing class.

- Extending an existing class with another.
- Implementing an interface in an existing class.
- Converting checked exceptions into unchecked exceptions.

Inter-type member declarations in AspectJ allow introduction of new members or constructors into classes or interfaces. The example in Figure 2.10 illustrates the introduction of a new private field named `identity`, two public methods named `getIdentity` and `setIdentity` and a new constructor to the class `Employee`.

```
aspect A {
private void Employee.identity=0;

public Employee.new(int id){
  identity=id;
}

public int Employee.getIdentity(){
  return identity;
}

public void Employee.setIdentity(int id){
  identity=id;
}

}
```

Figure 2.10: AspectJ Example2

AspectJ also allows us to change the inheritance hierarchy of existing classes using the construct `declare parents`. This construct can declare new super-classes or super-interfaces. The following example illustrated in Figure 2.11 shows that the class `Employee` becomes now a sub-class of a class named `Person` and implements the interface `Serializable`.

```
aspect ModifyEmployee {
declare parents: Employee extends Person;
declare parents: Employee implements Serializable;
...
}
```

Figure 2.11: AspectJ Example3

```
aspect A {
public void Play.run() {
...
}

declare parents: Play implements Runnable;
}
```

Figure 2.12: AspectJ Example4

The example in Figure 2.12 combines the inter-member declarations and the class hierarchy modification. The aspect `A` makes the class `Play` runnable. The aspect defines a void `run()` method for the class `Play` and declare it as `Runnable`.
In this thesis, we focuss only on AspectJ dynamic crosscutting.

## 2.4 Lambda Calculi and Type Systems

This section presents the background knowledge necessary for the reader to understand the AOP calculus λ_SAOP that we present in Chapter 6. It is also a state of the art of the most relevant contributions concerning type systems for lambda calculi. We describe first the type-free lambda calculus. Second, we present the simply typed lambda calculus. Finally, we give two different extensions of the simply typed lambda calculus, one with type schemes and the other with effects.

### 2.4.1 Type-Free Lambda Calculus

The lambda calculus [22] is a core language introduced by Alonzo Church in 1936 . The most important aspects of lambda calculus are the syntax of terms and the reduction (rewrite) relation on these terms. The language has a sparse syntax and a simple semantics. Lambda expressions, as shown in Figure 2.13, come in four varieties: Constants, Variables, Functions applications, and Lambda abstractions(functions definitions).

| | | | | | |
|---|---|---|---|---|---|
| *Exp* | ∋ | *e* | ::= | *c* | **Constant** |
| | | | \| | *x* | **Variable** |
| | | | \| | $e_1 e_2$ | **Application** |
| | | | \| | $\lambda x.e$ | **Abstraction** |
| *Const* | ∋ | *c* | ::= | `n \| ( ) \| true \| false` | |

Figure 2.13: Lambda Calculus Syntax

We present hereafter two examples of lambda expressions:

- $(\lambda x.x)$ describes the identity function since $((\lambda x.x)\ E) = E$ for any lambda expression $E$.
- $(\lambda f.(\lambda x.(f\ (f\ x))))$ denotes a function with two arguments: a function and a value that applies the function to the value twice.

The meaning of a lambda expression consists in the lambda expression that results after all its function applications are computed. The evaluation of a lambda expression is called reduction and it consists in substituting expressions for free variables. We define in the following paragraph the concepts of free and bound variable occurrences, free and bound variables, and the substitution of expressions for variables.

*- Free and Bound Occurrence*

An occurrence of a variable $v$ in a lambda expression is bound if and only if it is within the scope of a $\lambda$; otherwise it is called free.

- *Free and Bound Variable*

A variable is bound in a lambda expression if any of its occurrences are bound. A variable is free in a lambda expression if any of its occurrences are free.

A variable can be both bound and free in the same lambda expression. For example, in the lambda expression$((\lambda x.x)\ x)$ the two first occurrences of $x$ are bound whereas the third one is free.

- *Substitution*

The substitution of an expression $E'$ for a free variable $v$ in a lambda expression $E$ is denoted by $E[v/E']$ and is defined as follows:

1. $v[v/E'] = E'$ for any variable $v$
2. $x[v/E'] = x$ for any variable $x \neq v$
3. $c[v/E'] = c$ for any constant $c$
4. $(EE'')[v/E'] = (E[v/E'])(E''[v/E'])$
5. $(\lambda v.E)[v/E'] = (\lambda v.E)$
6. $(\lambda x.E)[v/E'] = \lambda x.(E[v/E'])$ when $x \neq v$ and $x$ is not free in $E'$
7. $(\lambda x.E)[v/E'] = \lambda z.(E[x/z][v/E'])$ when $x \neq v$ and $x$ is free in $E'$, where $z \neq v$ and $z$ is not free is $(EE')$

- *Example*

$(\lambda y.(\lambda f.(fx))y)[x/(fy)]$

$= \lambda z.((\lambda f.(fx))z)[x/(fy)]$ using 7

$= \lambda z.((\lambda f.(fx))[x/(fy)]z[x/(fy)])$ using 4

$= \lambda z.((\lambda f.(fx))[x/(fy)]z)$ using 2

$= \lambda z.((\lambda g.(gx)[x/(fy)])z)$ using 7

$$= \lambda z.((\lambda g.g(fy))z) \qquad \text{using 4, 2 and 1}$$

**Lambda Calculus Reduction**

The evaluation of a lambda expression involves the reduction of the expression until no more reduction rules can be applied. The main rule for simplifying a lambda expression, is called β-reduction. Another rule called α-reduction is used to rename bound variables in order to avoid variable capture when substituting free variables in an expression.

*- Alpha-Reduction*

The alpha-reduction (α-reduction) allows bound variable names to be changed. Name conflicts can be avoided in alpha-reduction if new variable names are used. For example, if $v$ and $w$ are variables, $E$ a lambda expression and $w$ does not appear in $E$, then:

$$\lambda v.E \underset{\alpha}{\rightarrow} \lambda w.E[v/w]$$

For example, $\lambda x.x$ can be alpha reduced to $\lambda y.y$.

*- Beta-Reduction*

The beta-reduction (β-reduction) expresses the idea of function application. The beta reduction of $((\lambda x.E)E')$ is simply $E[x/E']$ and this is denoted as follows:

$$(\lambda x.E)E' \underset{\beta}{\rightarrow} E[x/E']$$

Hence, $(\lambda x.E)E'$ is β-reduced to $E[x/E']$. If there are name clashes, alpha conversion may be required first.

**Beta-Normal Form**

A term M is said to be in beta-normal form (β-nf) if M has no part of the form $(\lambda x.M)N$. Such part is called β-redex.

*- Example*

$$((\lambda y.(\lambda f.(fx))y))(fy) \underset{\beta}{\rightarrow} \lambda z.((\lambda g.g(fy))z)$$

The expression $\lambda z.((\lambda g.g(fy))z)$ is in normal form and is the normal form of $((\lambda y.(\lambda f.(fx))y))(fy)$

### 2.4.2 Simply Typed Lambda Calculus

Typed lambda calculi are refinements of the untyped lambda calculus. They are foundations of typed functional programming languages such as ML [44] and Haskell [49]. There are two ways in which expressions can be typed: Implicit typing and explicit typing. The explicit typing, introduced by Church in [23], is based on a language of typed terms. In contrast, in Curry's type-theory [26] that introduced the implicit typing, the terms are untyped and formal rules assigned types to terms. In this thesis, we are interested by implicit typing systems and we present hereafter the simply typed lambda calculus in style of Curry and its different extensions. The readers interested by the area of explicit typing are invited to read [8,23].

Simply typed lambda calculus contains the types: *unit*, *bool*, *int*, and functional types. The type *unit* is the type with only one element *()* that allows typing functions without arguments or without return value. Assuming a set of type variables $V=\{\alpha,\beta,...\}$, the syntax of types $\tau$ is given in Figure 2.14. The symbol $\rightarrow$ associates to the right: we will read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ where $\tau_1$, $\tau_2$ and $\tau_3$ are given types.

$$\tau \quad ::= \quad unit \mid int \mid bool \mid \alpha \mid \tau \rightarrow \tau$$

Figure 2.14: Type Syntax

The typing inference rules are used to deduce typing statements, which associate types to lambda expressions. A typing statement that expresses that an expression *e* has a type $\tau$ under some typing environment $\Gamma$ is written:

$$\Gamma \vdash e : \tau$$

A typing environment $\Gamma$ is a map from variables to types and describes assumptions about variable types. It is denoted by $[x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n]$. The typing inference rules of simply typed lambda calculus need also the following notations:

- $\Gamma(x)$ denotes the type of $x$ in $\Gamma$, if such type exists.
- $\Gamma_x$ describes the map $\Gamma$ excluding the associations of the form $x \mapsto \_$.
- $\Gamma\dagger[x \mapsto \tau]$ denotes $\Gamma_x \cup \{x \mapsto \tau\}$

The typing inference rules are given in Figure 2.15. The function TypeOf used in the typing rule of a constant is given in Appendix III of the thesis and returns the type of the constant.

$$\frac{}{\Gamma \vdash c : \text{TypeOf}(c)} \quad \textbf{(T-const)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \textbf{(T-var)}$$

$$\frac{\Gamma\dagger[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad \textbf{(T-abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad \textbf{(T-app)}$$

Figure 2.15: Simply Typed Lambda Calculus Rules

### 2.4.3 Polymorphic Type System

In this section, we present another type system à la Curry where types are polymorphic. In the simply typed lambda calculus presented in the previous section, we can deduce, for any type $\sigma$, the following typing statement:

$$\Gamma \vdash \lambda x.x : \tau \rightarrow \tau$$

The type schemes introduced by Girard [43] and Reynolds [82] allow to express this fact with a unique notation:

$$\Gamma \vdash \lambda x.x : \forall\alpha.\alpha \rightarrow \alpha$$

Assuming a set of type variables $V=\{\alpha,\beta...\}$, the syntax of types $\tau$ and type schemes $\sigma$ is given in Figure 2.16.

| | | |
|---|---|---|
| $\tau$ | ::= | *unit* \| *int* \| *bool* \| $\alpha$ \| $\tau \rightarrow \tau$ |
| $\sigma$ | ::= | $\tau$ \| $\forall\alpha.\sigma$ |

Figure 2.16: Type and Type Scheme Syntax

A type scheme $\forall\alpha_1 \ldots \forall\alpha_n.\tau$ has generic type variables $\alpha_1, \ldots, \alpha_n$ and is denoted by $\forall\alpha_1 \ldots \alpha_n.\tau$. We present hereafter the type instantiation and the typing rules.

### Type Instantiation

A type $\tau$ has a type instance $\tau'$ if there exists a substitution $S$ such that $S\tau = \tau'$, where $S$ is a substitution of types for type variables, often written $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ or $[\tau_i/\alpha_i]$. $S\tau$ is the type obtained by replacing each occurrence of $\alpha_i$ in $\tau$ by $\tau_i$.

By contrast, a type scheme $\sigma=\forall\alpha_1 \ldots \alpha_m.\tau$ has a generic type instance $\tau'$ (written as $\sigma \succ \tau'$) if there exists a substitution $S$ such that $S\tau = \tau'$.

### Typing Rules

The typing rules in presence of type schemes are described in Figure 2.17.

### Example

$\Gamma \vdash \lambda xy.x : \forall\beta\alpha \rightarrow \beta \rightarrow \alpha$

$$\frac{\text{TypeOf}(c) \succ \tau}{\Gamma \vdash c : \tau} \quad \textbf{(T-const)}$$

$$\frac{x : \sigma \in \Gamma \quad \sigma \succ \tau}{\Gamma \vdash x : \tau} \quad \textbf{(T-var)}$$

$$\frac{\Gamma \dagger [x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad \textbf{(T-abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad \textbf{(T-app)}$$

Figure 2.17: Polymorphic Typed Lambda Calculus Rules

### 2.4.4 Effect-Based Type Systems

Lambda calculi extended with side effects need more sophisticated type systems and much investigations have been devoted to this issue [63, 94, 100]. Among other researchers, Gifford and Lucassen [42, 65], and pursued by Talpin and Jouvelot [89, 90], proposed an effect typing discipline to express computational effects within a program. In an effect-based type system, types describe what expressions compute whereas effects describe how expressions compute. In this section, we present an effect-based type system for an extended version of lambda calculus , where expressions, as shown in Figure 2.18, are:

- Constants
- Variables.
- Function abstraction.
- Function application.
- Let expressions.
- sequencing.

- Imperative notations such as referencing. An expression of the form `ref`$(e)$ allows the allocation of a new reference that points to the value obtained from the evaluation of $e$. The unary operator `!` is used for dereferencing, and the binary operator `:=` is used for assignment.

| | | | | | |
|---|---|---|---|---|---|
| *Exp* | ∋ | $e$ | ::= | $c$ | **Constant** |
| | | | \| | $x$ | **Variable** |
| | | | \| | $\lambda x.e$ | **Abstraction** |
| | | | \| | $e_1 e_2$ | **Application** |
| | | | \| | `let x =` $e_1$ `in` $e_2$ | **Let Expression** |
| | | | \| | `let rec` $f\ x = e_1$ `in` $e_2$ | **Let Rec Expression** |
| | | | \| | $e_1 ; e_2$ | **Sequencing** |
| | | | \| | `ref` $(e)$ | **Referencing** |
| | | | \| | `!` $e$ | **Dereferencing** |
| | | | \| | $e_1$ `:=` $e_2$ | **Assignment** |
| *Const* | ∋ | $c$ | ::= | `n \| ( ) \| true \| false` | |

Figure 2.18: Extended λ-Calculus Syntax

Types, effects and regions are used to control the type generalization in the presence of mutable data. Effects are used to describe the store whereas regions are intended to abstract memory locations. Types and effects in the extended lambda calculus are defined in Figure 2.19. The domain of regions $\rho$ is the disjoint union of a countable set of constants ranged over by $r$ and variables ranged over by $\gamma$. Basic effects $\eta$ can either be the constant $\emptyset$ that represents the absence of effects, effect variable $\varsigma$, $init(\rho, \tau)$ that stands for the allocation of a reference in a region $\rho$ to a value of type $\tau$, $read(\rho, \tau)$ that describes accesses to references in region $\rho$, or $write(\rho, \tau)$ that represents the assignments of values to references in the region $\rho$. The sequencing $\eta;\eta$'denotes the sequencing $\eta$ and $\eta$'. The type *unit* is the type with only one element *()* and $\alpha$ is a type variable. The term $ref_\rho(\tau)$ denotes reference types in a region $\rho$ to values of type $\tau$. The term $\tau \xrightarrow{\eta} \tau$' is the type of functions that take parameters of type $\tau$ to values of type $\tau$' with a latent effect $\eta$. By latent effect, we mean the effect generated when the corresponding function is evaluated [27].

$$
\begin{array}{llllllll}
\textit{Effect} & \ni \eta & ::= & \emptyset & | & \varsigma & | & \eta;\eta' \quad | \quad \textit{init}(\rho,\tau) \\
 & & & \textit{read}(\rho,\tau) & | & \textit{write}(\rho,\tau) & & \\
\textit{Type} & \ni \tau & ::= & \textit{unit} \quad | \quad \textit{int} \quad | \quad \textit{bool} \quad | \quad \alpha \quad | \quad \tau \xrightarrow{\eta} \tau' \quad | \quad \textit{ref}_{\rho}(\tau) & & & &
\end{array}
$$

Figure 2.19: Types and Effects in Extended λ-Calculus

**Typing Rules**

In the effect-based type system, the typing judgment is written $\Gamma \vdash e : \tau, \eta$ and states that expression $e$ has type $\tau$ and effect $\eta$ under some typing environment $\Gamma$. A type scheme is of the form $\forall \upsilon_1 ... \upsilon_n.\tau$ where $\upsilon_i$ can be type, region, or effect variable. A type $\tau'$ is a generic type instance of a type scheme $\sigma = \forall \upsilon_1 ... \upsilon_n.\tau$ (written as $\sigma \succ \tau'$) if there exists a substitution $S$ defined over $\upsilon_1 ... \upsilon_n$ such that $S\tau = \tau'$. Substitutions in this case map type variables to types and effect variables to effects.

We need also to define the notion of free variables and the notion of generalization to handle the typing of let expressions [27]. Type generalization states that a variable cannot be generalized if it is free in the typing environment $\Gamma$ or if it is present in the inferred effect. This is represented by the following function *Gen*:

$$Gen(\Gamma,\tau,\eta) = \forall \mathcal{F}(\tau) \backslash (\mathcal{F}(\Gamma) \cup \mathcal{F}(\eta)).\tau$$

where $\mathcal{F}(-)$ denotes the set of free variables as defined in the following:

$$
\begin{array}{lcl}
\mathcal{F}(\textit{unit}) & = & \{\,\} \\
\mathcal{F}(\alpha) & = & \{\alpha\} \\
\mathcal{F}(\tau_1 \xrightarrow{\eta} \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\tau_2) \cup \mathcal{F}(\eta) \\
\mathcal{F}(\textit{ref}_{\rho}(\tau)) & = & \mathcal{F}(\rho) \cup \mathcal{F}(\tau)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{F}(\forall \upsilon_1 ... \upsilon_n.\tau) & = & \mathcal{F}(\tau) \backslash \{\upsilon_1, ..., \upsilon_n\} \\
\mathcal{F}(\Gamma) & = & \bigcup_{x \in \mathrm{Dom}(\Gamma)} \mathcal{F}(\Gamma(x)) \\
\mathcal{F}(\emptyset) & = & \{\,\} \\
\mathcal{F}(\varsigma) & = & \{\varsigma\} \\
\mathcal{F}(\mathit{init}(\rho, \tau)) & = & \mathcal{F}(\rho) \cup \mathcal{F}(\tau) \\
\mathcal{F}(\mathit{read}(\rho, \tau)) & = & \mathcal{F}(\rho) \cup \mathcal{F}(\tau) \\
\mathcal{F}(\mathit{write}(\rho, \tau)) & = & \mathcal{F}(\rho) \cup \mathcal{F}(\tau) \\
\mathcal{F}(\eta; \eta') & = & \mathcal{F}(\eta) \cup \mathcal{F}(\eta') \\
\mathcal{F}(r) & = & \{\,\} \\
\mathcal{F}(\gamma) & = & \{\gamma\}
\end{array}
$$

Typing rules of the effect-based type system are given in Figure 2.20. The rules (**T-const**) and (**T-var**) manipulate type schemes by using the mechanism of generic instantiation. In the abstraction (**T-abs**), the effect of a lambda abstraction body is put inside the function while in the application rule (**T-app**) this embedded effect is extracted from the function type to be exercised at the point of call. Effects flow from the points where functions are defined to the points where they are used [89]. In the rules (**T-seq**), (**T-letrec**), and (**T-let**) sequencing of effects is used. The rule (**T-ref**) applies an effect for the allocation of a reference whereas (**T-deref**) and (**T-assign**) describe respectively access to a reference and assignment of a value to a reference.

### Inference Algorithm

The type and effect discipline that we adopt in the Chapter 6 is a variant of the one of Talpin and Jouvelot [89]. For this reason, we give in this section an overview of the type inference algorithm $\mathcal{J}$ of Talpin and Jouvelot. The type inference algorithm $\mathcal{J}$ is a constraint satisfaction problem that computes equalities between types and regions, and inequalities between effects. The effect-based inference algorithm takes as input a typing

$$\frac{\text{TypeOf}(c) \succ \tau}{\Gamma \vdash c : \tau, \emptyset} \quad \textbf{(T-const)}$$

$$\frac{x : \sigma \in \Gamma \quad \sigma \succ \tau}{\Gamma, s \vdash x : \tau, \emptyset} \quad \textbf{(T-var)}$$

$$\frac{\Gamma\dagger[x \mapsto \tau_1] \vdash e : \tau_2, \eta}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\eta} \tau_2, \emptyset} \quad \textbf{(T-abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\eta} \tau_2, \eta' \quad \Gamma \vdash e_2 : \tau_1, \eta''}{\Gamma \vdash e_1 e_2 : \tau_2, ((\eta;\eta');\eta'')} \quad \textbf{(T-app)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \eta \quad \Gamma \vdash e_2 : \tau_2, \eta'}{\Gamma \vdash e_1; e_2 : \tau_2, (\eta;\eta')} \quad \textbf{(T-seq)}$$

$$\frac{\begin{array}{c}\Gamma\dagger[x \mapsto \tau_1, f \mapsto \tau_1 \xrightarrow{\eta} \tau] \vdash e_1 : \tau, \eta \\ \Gamma\dagger[f \mapsto \mathit{Gen}(\Gamma, \tau_1 \xrightarrow{\eta} \tau, \eta)] \vdash e_2 : \tau_2, \eta'\end{array}}{\Gamma \vdash \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau_2, (\eta;\eta')} \quad \textbf{(T-letrec)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \eta \quad \Gamma\dagger[x \mapsto \mathit{Gen}(\Gamma, \tau_1, \eta)] \vdash e_2 : \tau_2, \eta'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, (\eta;\eta')} \quad \textbf{(T-let)}$$

$$\frac{\Gamma \vdash e : \tau, \eta}{\Gamma \vdash \texttt{ref}\ (e) : \mathit{ref}_\rho\ (\tau), (\eta; \mathit{init}(\rho, \tau))} \quad \textbf{(T-ref)}$$

$$\frac{\Gamma \vdash e : \mathit{ref}_\rho\ (\tau), \eta}{\Gamma \vdash\ !e : \tau, (\eta; \mathit{read}(\rho, \tau))} \quad \textbf{(T-deref)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{ref}_\rho\ (\tau_1), \eta \quad \Gamma \vdash e_2 : \tau_1, \eta'}{\Gamma \vdash e_1 := e_2 : \mathit{unit}, ((\eta;\eta'); \mathit{write}(\rho, \tau_1))} \quad \textbf{(T-assign)}$$

Figure 2.20: Typing Rules with Effects

environment $\Gamma$ and an expression *e*. The algorithm either fails or terminates successfully producing a 4-tuple whose components are: a substitution $\theta$, a type $\tau$, an effect $\eta$, and a set of constraints *k*. The substitution records those substitutions that take place during the various recursive calls of the inference algorithm and that range over the free variables of the environment $\Gamma$. The type produced by the algorithm is the inferred type of the argument expression. The effect produced by the algorithm corresponds to the minimal approximation of the effects that may be generated when the expression is evaluated. The constraint set *k* consists of inequalities between effect variables and effect sets. The inequality $\eta \subseteq \varsigma$ in *k* enforces a lower bound $\eta$ for the inferred effect variable $\varsigma$ consistent with the static semantics. Constraints are built during the processing of lambda and rec expressions, which is the place where effects are introduced into types.

A substitution $\bar{k}$ from effect variables to effects is a model of a constraint set *k*, if and only if, for each inequality $\eta \subseteq \varsigma \in k$, $\bar{k}\eta \subseteq \bar{k}\varsigma$. By construction, constraint sets always admit at least one solution. In the static semantics, type schemes are of the form $\forall \upsilon_1 ... \upsilon_n.\tau$ where each $\upsilon_i$ can be a type, region, or effect variable. In the algorithm, effects are represented by variables and are determined by a set of constraints. Consequently, type schemes will now be of the form $\forall \upsilon_1 ... \upsilon_n.(\tau, k)$ where *k* is a set of constraints.

The type algorithm applies an algorithm *CTypeOf* to type a constant expression. A unification algorithm $\mathcal{U}$ is also used to solve the equations on types, regions, and effect variables that are built by the algorithm $\mathcal{J}$. It returns a substitution $\theta$ as the most general unifier of two terms, or fails. The algorithms $\mathcal{U}$ and *CTypeOf* are described hereafter:

- The Algorithm *CTypeOf*

*CTypeOf*$(c)$= **case** $(c)$ **of**

| | |
|---|---|
| *n* | $\Rightarrow (int, \{\})$ |
| ( ) | $\Rightarrow (unit, \{\})$ |
| `true` | $\Rightarrow (bool, \{\})$ |
| `false` | $\Rightarrow (bool, \{\})$ |

- The Unification Algorithm $\mathcal{U}$

$$
\begin{array}{ll}
\mathcal{U}(\tau,\tau') = \textbf{case } (\tau,\tau') \textbf{ of} & \\
(\alpha,\alpha') & \Rightarrow [\alpha \mapsto \alpha'] \\
(\alpha,\tau) \mid (\tau,\alpha) & \Rightarrow \textbf{if } \alpha \in \mathcal{F}(\tau) \\
 & \quad \textbf{then } \mathit{fail} \\
 & \quad \textbf{else } [\alpha \mapsto \tau] \\
(\tau_i \xrightarrow{\varsigma} \tau_f, \tau'_i \xrightarrow{\varsigma'} \tau'_f) & \Rightarrow \textbf{let } \theta_1 = [\varsigma \mapsto \varsigma'] \\
 & \quad\quad \theta_2 = \mathcal{U}(\theta_1\tau_i, \theta_1\tau'_i) \\
 & \quad \textbf{in } \mathcal{U}(\theta_2\theta_1\tau_f, \theta_2\theta_1\tau'_f)\theta_2\theta_1 \\
(\mathit{ref}_\gamma(\tau), \mathit{ref}_{\gamma'}(\tau')) & \Rightarrow \textbf{let } \theta = [\gamma \mapsto \gamma'] \textbf{ in } \mathcal{U}(\theta\tau, \theta\tau')\theta \\
(\mathit{unit}, \mathit{unit}) & \Rightarrow \mathit{id} \\
(\mathit{int}, \mathit{int}) & \Rightarrow \mathit{id} \\
(\mathit{bool}, \mathit{bool}) & \Rightarrow \mathit{id} \\
\textbf{else} & \Rightarrow \mathit{fail}
\end{array}
$$

To define the generalization function employed by the type inference algorithm, we need the following two auxiliary definitions. First, a set $Y$ of variables is said to respect a set $k$ of constraints if each constraint of $k$ satisfies that it either only involves variables of $Y$ or of the complement of $Y$. Second, a closure of a set $X$ under $k$ is defined by the formula:

$$X^k = \{v_n | v_0 \in X \wedge \forall i < n : (...v_{i+1}... \leq v_i) \in k\}$$

The generalization function is defined then as follows:

$$\mathit{Gen}_k(\Gamma,\tau,\eta) = \forall \mathcal{G}(\Gamma,k,\tau,\eta)(\tau,k) \text{ where}$$
$$\mathcal{G}(\Gamma,k,\tau,\eta) = \bigcup\{V | V \subseteq \mathcal{F}(\tau)^k \backslash (\mathcal{F}(\Gamma) \cup \mathcal{F}(\eta)), V \text{ respects } k\}$$

In the following, we present the inference algorithm $\mathcal{J}$.

$\mathcal{J}(\Gamma, c) =$

**let** $\forall \upsilon_1, \ldots, \upsilon_n(\tau, k) = \mathit{CTypeOf}(c)$

$\upsilon'_1, \ldots, \upsilon'_n$ **new**, $\theta = [\upsilon_1 \mapsto \upsilon'_1, \ldots, \upsilon_n \mapsto \upsilon'_n]$

**in** $(id, \theta\tau, \emptyset, \theta k)$

$\mathcal{J}(\Gamma, x) =$

**if** $x \notin \mathrm{Dom}(\Gamma)$ **then** *fail*

**else**

**let** $\forall \upsilon_1, \ldots, \upsilon_n(\tau, k) = \Gamma(x)$

$\upsilon'_1, \ldots, \upsilon'_n$ **new**, $\theta = [\upsilon_1 \mapsto \upsilon'_1, \ldots, \upsilon_n \mapsto \upsilon'_n]$

**in** $(id, \theta\tau, \emptyset, \theta k)$

$\mathcal{J}(\Gamma, \lambda x.e) =$

**let** $\alpha, \varsigma$ **new**

$(\theta, \tau, \eta, k) = \mathcal{J}(\Gamma\dagger[x \mapsto \alpha], e)$

**in** $(\theta, \theta\alpha \xrightarrow{\varsigma} \tau, \emptyset, k \cup \{\eta \subseteq \varsigma\})$

$\mathcal{J}(\Gamma, e_1 e_2) =$

**let** $(\theta_1, \tau_1, \eta_1, k_1) = \mathcal{J}(\Gamma, e_1)$

$(\theta_2, \tau_2, \eta_2, k_2) = \mathcal{J}(\theta_1\Gamma, e_2)$

$\alpha, \varsigma$ **new**, $\theta_3 = \mathcal{U}(\theta_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha)$

**in** $(\theta_3\theta_2\theta_1, \theta_3\alpha, \theta_3(\theta_2\eta_1; \eta_2; \varsigma), \theta_3(\theta_2 k_1 \cup k_2))$

$\mathcal{J}(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) =$

**let** $(\theta_1, \tau_1, \eta_1, k_1) = \mathcal{J}(\Gamma, e_1)$

$(\theta_2, \tau_2, \eta_2, k_2) =$

$\mathcal{J}(\theta_1\Gamma\dagger[x \mapsto Gen_{k_1}(\theta_1\Gamma, \tau_1, \eta_1)], e_2)$

**in** $(\theta_2\theta_1, \tau_2, \theta_2\eta_1; \eta_2, k_2)$

$\mathcal{J}(\Gamma, \textbf{ref}\ e) =$

**let** $\gamma$ **new**

$(\theta, \tau, \eta, k) = \mathcal{J}(\Gamma, e)$

**in** $(\theta, ref_\gamma(\tau), \eta; init(\gamma, \tau), k)$

$\mathcal{J}(\Gamma, !e) =$

**let** $(\theta_1, \tau_1, \eta_1, k_1) = \mathcal{J}(\Gamma, e)$

$\alpha, \varsigma$ **new**, $\theta_2 = \mathcal{U}(ref_\gamma(\alpha), \tau_1)$

**in** $(\theta_2\theta_1, \theta_2\alpha, \eta_1; read(\theta_2\gamma, \theta_2\alpha), \theta_2 k_1)$

$\mathcal{J}(\Gamma, e_1 := e_2) =$

**let** $(\theta_1, \tau_1, \eta_1, k_1) = \mathcal{J}(\Gamma, e_1)$

$(\theta_2, \tau_2, \eta_2, k_2) = \mathcal{J}(\theta_1\Gamma, e_2)$

$\gamma$ **new**

$\theta_3 = \mathcal{U}(ref_\gamma(\tau_2), \theta_2\tau_1)$

**in** $(\theta_3\theta_2\theta_1, unit, \theta_3(\theta_2\eta_1; \eta_2; write(\gamma, \tau_2)),$

$\theta_3(\theta_2 k_1 \cup k_2))$

Finally, if $\mathcal{J}(\Gamma, e) = (\theta, \tau, \eta, k)$ then $\bar{k}\theta\Gamma \vdash e : \bar{k}\tau, \bar{k}\eta$.

## 2.5 Research Initiatives

In this section, we present the main contributions related to the topic of the thesis. We will cite the contributions where AOP is used for security goals then the contributions that formalize the semantics of AOP languages followed by those that describe semantics for

JVML.

### 2.5.1 AOP and Security

AOP is a very promising paradigm for software security. Among the attempts that have been made to use AOP for security, we can cite the DARPA-Funded project of Cigital Labs [59, 83, 96] that applies AOP to enforce secure code practices. The main outcomes of this project are a security dedicated aspect extension of C called CSAW [59] and a weaving tool. The operations handled by a CSAW aspect are the same as the other AOP languages: replace code or insert code before or after the points of interest. In addition to the standard calls to functions, CSAW defines new points of interests: function definitions, a line of a code following a label and a block code between two labels. However, CSAW addresses local, small-sized problems as buffer overflows and data logging.

De Win et al. [29,31,33,95,99] explored the use of AOP to integrate security aspects within applications. In [32], De Win et al. applied AspectJ to enforce access control and in his thesis [30], De Win modularized the auditing and access control features of the FTP server and is then able to run the server with or without security depending on whether the security aspects are weaved into the system or not. In his thesis, he also criticized the pointcut-advice model of AspectJ and made the case for dedicated aspect languages for security. However none of his works presents a new security aspect oriented language.

Another contribution is the security aspect library JSAL [48], which is implemented in AspectJ and provides security functions. It is based on the Java security packages JCE [70] and JAAS [60]. The implementation leverages the abstract pointcuts of AspectJ in order to reuse aspects. This is, however, a very limited library that shows the feasibility of a framework based on the AOP paradigm for the reuse and integration of pre-built security aspects.

In [15], Ron Bodkin describes examples of security crosscutting that are frequently encountered. In this paper, he analyzes the relevant join points and properties that can be used in AOP systems and identifies areas of future work. However, this contribution

targets only Web applications and does not implement any of the proposed ideas in an AOP language.

There is another kind of contribution in the field of AOP for enriching the expressiveness of pointcuts for some goals. In a security perspective, Masuhara and Kawauchi [53, 66] present a new pointcut called dataflow pointcut. They show in their papers, that some security concerns, such as secrecy and integrity, are sensitive to flow of information in a program execution. The data flow pointcut identifies join points based on the origins of values, and can be used with the other kinds of pointcuts in existing AOP languages. However the prototype implementation is done on a simple,pure object-oriented language with a pointcut-and-advice mechanism and not on a fully fledged language as AspectJ.

### 2.5.2 Formal Semantics for AOP

A number of design and implementation efforts have been proposed for aspect oriented languages. But to date, there are only few contributions that formalize features of aspect oriented languages and give formal semantics. A criterion by which we can classify the state of the art on AOP languages semantics is the description style of the semantics: Denotational, operational or axiomatic [72]. In an operational semantics the meaning of a program is defined by specifying the behavior of the program's execution, in a denotational semantics, meaning is defined abstractly via elements of a mathematical structure, called denotations, and in an axiomatic semantics, meaning is defined using some logic asserting properties. Two popular styles of operational semantics exist: big step semantics and small step semantics. The small step semantics precisely models implementations as a sequence of simple operations and it can take many steps to fully evaluate a program whereas the big step semantics is more abstract and does not take into account intermediate steps. Most formal works are described using small step semantics. Few contributions express semantics differently: big step [61], denotational [98] and as far as we have checked there is no axiomatic semantics for AOP.

Lämmel [61] extends a small Java-like object-oriented language called $\mu$O2 to incorporate a new construct `superimpose`, which allows the definition of an advice intercepting a method. He presents a big step operational semantics for this language. The construct `superimpose` enables a programmer to attach additional functionality to certain join points along the execution of specified method calls. Method call interceptors can be activated at arbitrary points in the control flow of a program. However, the operations that can be intercepted in such a language are only method calls, which is too restrictive. The focus on method calls only is not sufficient for most aspect languages.

Wand, Kiczales and Dutchyn [98] present a denotational semantics for pointcuts and advice for a procedural version of the pointcut-advice language of the British Columbia University project: Aspect Sand Box (ASB) [37]. ASB consists of a Scheme interpreter for a simple OO language, and several extensions modelling different AOP styles, including the pointcut-advice model of AspectJ. The languages used in ASB are simplified languages and contain features to characterize complex languages as AspectJ, HyperJ and DJ. The authors [98] have simplified more the ASB pointcut-advice language by removing types, classes, and objects from the language and by slightly simplifying the join point model. The semantics models essential characteristics of AspectJ as dynamic join points, pointcut designators and advices. However, several AspectJ characteristics have been left out. The mini-language of Wand et al. semantics considers, for example, only three kinds of join points: method call, method execution, and advice execution, whereas AspectJ counts eleven join points.

Andrews [7] presents a syntax and an operational semantics of a process algebra that he proposes as a possible foundation of AOP. The language is based on a subset of CSP algebra with prefixing, synchronization on a set, and external choice. The proposed process algebra has the characteristics of aspect oriented languages: A definition of join points (as points in the program text), a means of designing join points (by procedure names), and a means of affecting them with advices [7]. However, the language on which

is built this semantics is far from AspectJ syntax and cannot serve to understand the weaving mechanism in AspectJ, which contains different join points and pointcuts.

Jagadeesan, Jeffrey and Riely [52] present an operational semantics for an untyped base language with multithreading, classes and objects. They then enrich the syntax of the base language to handle aspects. They also give a translation from the language with advice to an equivalent language without advice, and show that the translation preserves the operational semantics. However, in this contribution, the semantics is done for a small calculus that considers only method call and method execution pointcuts.

Walker, Zdancewic and Ligatti [97] present an operational semantics for a simply-typed lambda calculus extended with two central new abstractions: explicitly labeled program points and advices. The labels serve both to trigger advice and to mark continuations that the advice may return to. The system is not intended to directly model constructs like AspectJ but is a calculus into which source-level AOP constructs can be translated. It could be considered more general than existing AOP languages but cannot be considered as a foundations for AspectJ semantics.

Douence, Motelet and Sudholt present an operational semantics of an AOP system [35]. They describe a domain specific language for the definition of crosscuts and the system is based on a monitor that observes the behavior of the programs. The monitored program calls the monitor when an event is emitted and the monitor will then check if there is any crosscut at this point. In case of crosscutting, the monitor will perform the respective action by replacing the call to the original method by a call to another one. Otherwise, it will call the original method. This approach is called EAOP for Event Aspect oriented Programming. The framework is based on the following simple principles: join points are modelled as events, pointcuts are specified as patterns of event sequences, and advices are executed when an execution trace of the program matches their pointcuts. However, the contribution of Douence et al. is more geared towards a formal understanding and less towards a semantics of a fully fledged language.

The following contributions are close to semantics ones. Masuhara and Kiczales [67] present a single framework to define the core semantics of four aspect oriented languages, among them languages based on the pointcut-advice model as in AspectJ, multiple separation of concerns as in HyperJ and traversal specification as in DJ. A Scheme implementation for the interpreters represents an operational semantics of the languages; however, giving a semantics of a language through its interpreter is not appropriate for formal study.

Masuhara, Kiczales and Dutchyn [68] presented another work based on the ASB project that formalizes the use of aspects. Based on the operational semantics model of the language PA, the pointcut-advice language of ASB, and using partial evaluation, the paper shows how to compile PA into Scheme by partially evaluating the interpreter. However, this contribution does not define a formal semantics for the language.

In contrast to all this research, our aim is to develop first a formal semantics for AspectJ and secondly to develop a semantics for a security aspect oriented calculus with dedicated join points and pointcuts.

### 2.5.3 Formal Semantics for JVML

The most known AOP language, AspectJ [56], since the version 1.1, has been implemented using bytecode (JVML) weaving, which combines aspects and classes to produce .class files that run in a Java VM [38]. In order to understand how the weaving is done in AspectJ, a first step is to understand the semantics of Java and more precisely the semantics of the JVML.

The semantics of Java has been a fruitful area of research. Several proposals have been advanced. We can structure the related work (state of the art) on Java semantics into 2 categories: Semantics of the Java language and semantics of JVML. Among the most prominent proposals for Java semantics, we can cite: [6,18,20,21,36,50,51,73,74,74,88] and among the most prominent proposals on JVML semantics are: [12–14,17,25,39–41, 45,58,62,84,85,87].

Another criterion by which we can describe the state of the art on Java semantics is the description style of the semantics: Denotational, Operational or Axiomatic [72].

The following proposals adopt an operational approach to describe Java language or JVML semantics [12–14, 17, 21, 36, 39–41, 45, 58, 62, 73, 84, 85, 87, 88] whereas [6, 20, 51] present a denotational one. An axiomatic semantics is given in [74] for the Java Langage. Most of the research initiatives describing the semantics of JVML subsets use a small-step operational semantics [12–14, 39–41, 45, 58, 62, 84, 85, 87]. The main objective of these proposals is either to purely study the semantics or put focus more on typing constraints.

In his papers [12, 13], Bertelsen presents a very detailed semantics but does not address the semantics of multithreading and synchronization for `monitorenter` and `monitorexit` instructions. Freund and Mitchell use a type system to investigate the problem of object initialization and subroutines in [40] and add objects, classes, interfaces, arrays, exceptions in [39, 41]; however all these papers describe only the semantics of a single thread of execution and the JVM is then viewed as a single-threaded state machine. Hagiya and Tozawa in [45] and Klein and Wildmoser in [58] define operational semantics for simple languages with subroutines, along with a notion of type safety; however, in these proposals, method invocation, exceptions handling and multithreading are not taken into account.

In their technical report [14], Bigliardi and Laneve isolate a sublanguage of the JVM with thread creation and mutual exclusion and define an operational semantics and a formal verifier that enforces basic properties of threads, lock and unlock operations. In their work, they did not handle modifiers and did not consider the instructions `invokespecial`, `invokestatic` and `invokeinterface`. They give only a simple semantics for `invokevirtual` in case of void methods without arguments. Furthermore, they did not address the subtlety between creation of threads by extending the class `Thread` or by implementing the interface `Runnable`. In fact, they use an instruction `start`($\sigma$) as an artifact to the instruction `invokevirtual java/lang/`

`Thread/start()`. In another similar paper of Laneve [62], the invoke method instuctions have not been totally taken into account. Siveroni [84, 85] presents an operational semantics for a language that models the Java Card Virtual Machine including exception handling, array objects and subroutines but missing the multithreading aspect. Stata and Abadi [87] propose the use of typing rules for bytecode verification focussing more on subroutines and proved its soundness. They define an operational semantics for a JVML subset containing only 9 instructions `inc`, `pop`, `push`, `load`, `store`, `if`, `jsr`, `ret` and `halt` because the authors were mainly interested in addressing the problems caused by the subroutines. Börger and Schulte [17] have used operational semantics formalism of Abstract State Machines (ASMs) to describe the JVM with the goal of defining a platform for correct compilation of Java code.

None of the previously proposed operational semantics for JVML handle and describe in a detailed way method invocation instructions, modifiers, multithreading, and synchronization.

## 2.6 Conclusion

In this chapter we presented the basic principles of software security. We also provided an overview of the most important AOP approaches: Pointcut-advice model, multidimensional separation of concerns, and adaptive Programming. Particularly, we gave a detailed description of the most popular AOP language: AspectJ, which is the AOP language used along this thesis. Furthermore, we described the most important notions in lambda calculi and type systems. For the state of the art, we presented the contributions related to the use of AOP for security and related to the semantics foundations of AOP. The next chapter is devoted to a security appropriateness analysis of the AOP approaches and more precisely to an AspectJ security appropriateness analysis.

# Chapter 3

# Appropriateness Analysis of AOP for Security

The main objective of this chapter is to present a security appropriateness analysis for the AOP approaches mentioned in Section 2.2. We will show that the pointcut-advice model is the most interesting model for software security hardening. Because we are interested in security hardening of Java applications, we present a security analysis of AspectJ, which is the most popular AOP extension of Java. We finally cite the shortcomings of AspectJ and propose some extensions.

## 3.1 AOP approaches and Security

The fundamental concepts of the AOP approaches presented in Section 2.2 are different. The pointcut-advice approach is based on the notions of: join points, pointcuts, and advices. Multi-dimensional separation of concerns (MDSOC) allows developers to partition overlapping concerns in software along multiple dimensions of composition and decomposition. This approach is called symmetric because all concerns (base application and aspects) in the system are equally created and can be combined, as opposed to the

pointcut-advice approach where aspects are composed into the base application. Adaptive programming uses traversal strategies and adaptive visitors to implement crosscutting concerns.

From a security point of view, the multi-dimensional separation of concerns has a serious limitation. It does not allow to add functionality before, after, or around a field access. Access authentication to a given field in a given class is a simple security example that we cannot handle with HyperJ, which is a representative for the MDSOC model. The latter approach works at the method level and consequently cannot operate within a method body. HyperJ, for example, does not support pulling a part of code within method bodies. Picking out multiple concerns within method bodies is required in many situations to enforce security.

The adaptive programming approach is concerned with the loose coupling between structure and behavior and focuses on certain kinds of concerns. For example, DJ is unable to change a method by a more secure one.

The pointcut-advice model is the most popular model. It offers a better granularity than the MDSOC approach and considers more general kinds of concerns than adaptive programming. Furthermore, the pointcut-advice model adapts extensively the pull approach. It allows tracking subtle points in the control flow of the application. For example points where methods are invoked and fields are set. For these reasons, we adopt the pointcut-advice model for security hardening of applications. More precisely, we choose AspectJ as the candidate to enforce security issues in Java applications. In the sequel, we describe some extensions to AspectJ for security hardening of Java applications.

## 3.2 Suggested AspectJ Extensions

In this section, we identify the security issues that cannot be handled by AspectJ. Then, we suggest the extensions that should be added to AspectJ in order to successfully handle those security issues.

### 3.2.1 Predicted Control Flow Pointcut

Predicted control flow pointcuts identify join points based on the predicted behavior at the current join point. Hence, a pointcut `pcflow(p)` matches at a join point if there may exist a path to another join point where `p` matches. The idea was originally proposed by Kiczales [54], however it has never been concretized in AspectJ. Kiczales has discussed this new pointcut with the example that we describe in Figure 3.1 and Figure 3.2. The three classes in Figure 3.1 represent figures and the aspect in Figure 3.2 updates a display whenever a program changes any visual property of a figure. The definition of the class `Display` is omitted here. A `Display` object has a list of figures shown in it and we should ensure that it is updated when the state of its figure elements changes.

The *displayState* pointcut represents field gets under the predicted control of the draw methods. The pointcut *set(<displayState()>)* in the advice declaration matches all sets to the fields represented by *displayState*. The aspect calls *Display.update()* to redraw the modified figure at each time a program modifies a visual property (e.g., x of a Point object). Hence, the Pcflow pointcut means: (1) predict the control flow of all draw methods in the subclasses of `Fig`, (2) retrieve the set of fields that are read and represented by *<displayState()>*, and (3) update the display (with the advice) at any join point where a field contained in *<displayState()>* is modified.

We learn from this example how to harden the security of applications. In [93], the authors proposed a technique to detect intruders with visual data analysis. Based on this idea, we can draw some charts for security parameters such as file activity, registry activity, or network traffic. These charts can be analyzed to discover if something wrong happens. By using the same concept in Kiczales's example, any changes in these charts by setting these parameters in a way or another will not only be reflected in the display but also some necessary steps could be taken in response to such changes to protect the system. So Kiczales's example can be rewritten as in Figure 3.3.

```
abstract class Fig {
   abstract void draw ();
}

class Point extends Fig {
   int x,y;
   void draw () {
   Display . plotXY(x,y);
   }
}

class Line extends Fig {
   Point p1 ,p2;
   void draw () {
   Display.line (p1.x , p1.y, 14 p2.x , p2.y);
   }
}
```

Figure 3.1: Figure Classes for Pcflow

```
aspect DisplayUpdating {
   pointcut* displayState(): pcflow(execution(void Fig+.draw()))
                             && get(* Fig+.*);
   after set(<displayState()>)(): {Display.update(); }

}
```

Figure 3.2: Display Updating Aspect with pcflow

```
pointcut* displayState(): pcflow(void SecurityElement+.draw()))
                          && get(* SecurityElement+.*);
after set(<displayState()>)(): {Display.update(); }
```

Figure 3.3: Pcflow Pointcut Security Example

### 3.2.2 Dataflow Pointcut

Masuhara and Kawauchi [66] have defined a dataflow pointcut for security purposes that is not implemented yet in AspectJ. The pointcut identifies join points based on the origins of values. Cross-site scripting (XSS) problem in web-applications is an example that shows the need for such a pointcut. A Web site might be vulnerable to XSS attacks if it reflects

input back to the user such as search engines and shopping sites. Attacker crafts a link containing malicious code and let the victim click on it by different ways. The vicitm's browser transmits the attacker's code to the Web site as part of the URL. The Web site reflects the input to the victim's browser. The malicious code runs on the victim's browser because it thinks the code comes from the vulnerable Web site. We explain this kind of attacks with the example presented in Figure 3.4.

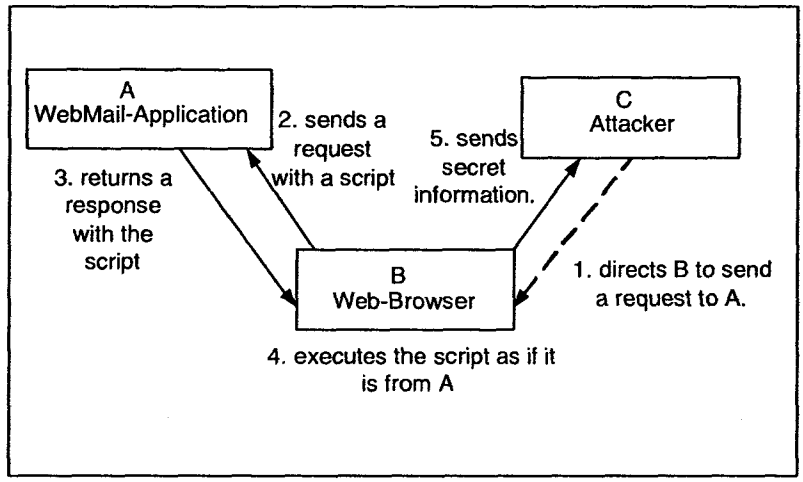



Figure 3.4: Cross Site Scripting Problem

By using the XSS attack on a vulnerable mail site, an attacker can access the e-mail account of a legitimate user, views the victim's messages and sends messages from the account. This attack presents privacy and non-repudiation risks. Here is a possible scenario.

1. *C* sends to *B* a document with a link to the login page of *A* with a script embedded as a parameter to the ID.

2. *B* follows the link and goes to the login page of *A*.

3. *A* returns a web page as a response to *B*'s request indicating a login failure because the script is not a valid username. Since *A* has a cross site scripting problem, the script is sent in the response.

4. *B* executes the malicious script in the returned page with the privileges of *A*.

5. The secret information of *B* that should only be accessed by *A* is sent to *C*.

Since scripts contain special characters, the crosscutting problem could have been avoided if *A* sanitized the data coming from untrusted sources by removing any special characters not intended to be there. In [66], the authors shown that with existing kinds of pointcuts, we can only sanitize all strings to be sent even they did not come from the user's input. In order to intercept only the strings coming from a return value of *getParameter* method, there is a need for a new pointcut based on the origin of the values. For this purpose, the dataflow pointcut has been introduced.

Here is another example that clarifies the need for the data flow pointcut. Assume that a program opens a confidential file, reads data from this file, and then sends data over the net. This is critical from a security point of view. A data flow analysis using a data flow pointcut can indicate whether the data sent over the net actually depends on the information read from the confidential file.

### 3.2.3 Loop Pointcut

Harbulot and Gurd present in [46] a loop join point model that demonstrates the need for a more complex join point in AspectJ. Their approach to recognize loops is based on a control-flow analysis at the bytecode level. They restricted their study to loops iterating over an iterator or a range of integers. This research lacks the analysis of infinite loops and loops that contain boolean conditions. Through pointcuts that pick out such loops, an excessive security problems can be solved easily.

An infinite loop is a set of instructions that executed repeatedly. This is a desired behavior in some situations as in database servers. Database servers loop forever waiting for a request to process it. However, infinite loop might also be bugs unintentionally made by programmers. Malicious code writers exploit infinite loops to do their nefarious

jobs by launching denial of service attacks. Denial of service attacks consume system resources until the application or the entire system becomes unusable.

Halting the web browser by running a code that opens a dialog window an infinite number of times is a denial of service attack. This attack requires rebooting the workstation. There is no general methods to specify whether a code will ever halt or run forever but AspectJ must include mechanisms to predict the existence of such infinite loops and then notify the user if she wants to continue with this work or not. As a suggestion, it is possible to add a pointcut that is associated with the loop body. Through an after advice with such a pointcut , we can increment a counter every time this body is executed. If the counter value becomes more than a threshold specified according to the type of the application, an alert is popped up giving the user the ability to abort the execution.

### 3.2.4 Pattern Matching Wildcard

There is a need for a new wildcard in AspectJ to perform pattern matching. Although pattern matching can be done by plain AspectJ, it is however better to do it in a declarative manner to simplify the code. We illustrate this point with an example related to security. Viruses always inject themselves inside executable files. So, it is essential to control opening and writing files that have an "`exe`" extension. For example, let us write a pointcut that picks out all constructor call join points of the form `FileWriter(x,y)` where the parameter `x` is a string whose value ends with the word "`exe`". Using plain AspectJ, the pointcut will have the following form:

```
    call (FileWriter.new(String,String)) && args(x,*) &&
if (isExeExtension(x));
```

where `isExeExtension` is a boolean method that tests if its argument value ends with the word "`exe`". Although we are able to write the pointcut using plain AspectJ, this has been done with an extra method like `isExeExtension`. We suggest another way that uses the keyword `like` and the `%` character of SQL. This will ease the burden on the user and simplify the code. The previous pointcut definition can be rewritten according to

our suggestion as:

```
call (FileWriter.new(String like "%exe%", String))
```

Obviously, using such wildcards states directly the programmer's intent and makes the program clear.

### 3.2.5 Type Pattern Modifiers

AspectJ uses, as described in Table 2.2 and Table 2.3, four kinds of patterns in the pointcut syntax: Method pattern, constructor pattern, field pattern, and type pattern. Patterns are used inside primitive pointcut designators to match signatures and consequently to determine the required join points. The syntax of these patterns as described in [79] contains the modifiers option except for the type pattern syntax. This section discusses the need for adding modifiers in the type pattern syntax.

A Java class declaration may include the following modifier patterns: public, abstract, or final. A public class is a class that can be accessed from other packages. An abstract class is a class that has at least one abstract method that is not implemented. A class that is declared as final may not be extended by subclasses. Any class, method, object, or variable that is not private is a potential entry point for an attack. Hence, using modifiers in the type pattern syntax should be very useful from a security point of view. The example in Figure 3.5 describes a case where the public method `f()` inside the public class `Sensitive` delivers sensitive information. In this case, it is essential to add a security mechanism that authenticates the clients of such public classes that are exposed by the application to the outside world. Hence, we would like to be able to use a `public` modifier pattern in type pattern syntax to pick out public classes only.

```
public class Sensitive {
   private String sensitiveInfo;
   public void f(){
     //...
     System.out.println(sensitiveInfo)
   }
}
```

Figure 3.5: Type Pattern Modifiers.

### 3.2.6 Local Variables

AspectJ allows to pick out join points where attributes are referenced or assigned through `get` and `set` designators but it does not provide similar pointcuts to local variables defined inside methods. New pointcut designators that do such a behavior will increase the efficiency of AspectJ especially from a security point of view. For example, security debuggers may need to track the values of local variables inside methods. With such new pointcuts, it will be easy to write advices before or after the use of these variables to expose their values. Confidential data can be protected using these kinds of pointcuts by preventing them from being used improperly. A promising approach [71] for protecting privacy and integrity of sensitive data is to statically check information flow within programs. Instead of doing static analysis, we propose to use AOP to insert checks before or after getting or setting fields or local variables. The following example in Figure 3.6 clarifies this idea.

The sensitive information stored in the private field `sensitiveInfo` has been exposed by transferring its value to the local variable `localstr` defined inside the method `f()`. Then, the value of `localstr` is stored inside the public field `publicInfo`, which made the information accessible from outside the class. Using pointcuts that track fields as well as local variables can help us to find such a case and prevent it.

```
class Test {
   private String sensitiveInfo;
   public String publicInfo;
   private void f(){
     String localstr;
     sensitiveInfo=/* Some Calculation*/;
     localstr=sensitiveinfo;
     //...
     publicInfo=localstr;
     //...
   }
}
```

Figure 3.6: Local Variables Get and Set.

### 3.2.7 Synchronized Block Joinpoint

The synchronized block has not been treated yet in AspectJ or in any other AOP framework. There are no join points associated with such a block so far. The current implementation of AspectJ allows picking out calls to synchronized methods but does not allow picking out synchronized blocks. The importance of the join points for synchronized code has been already discussed for thread management. Borner has presented a paper [19] on these issues and has discussed the usefulness of capturing synchronized blocks such as calculating the time to acquire or to release the lock. In this section, we do care about the importance of such pointcuts for security issues. Suppose we have a synchronized block that launches a denial of service attack by containing a code that eats the CPU cycles like the code that implements Ackerman function in [69]. Ackerman function is a function of two parameters whose value grows very fast. It is essential to have a join point at the beginning of the synchronized block. Through this join point, we can write a before advice that limits the CPU usage or limits the number of instructions that can be run. This limitation will counter the attack.

Let us take another example that we present in Figure 3.7.

We need to insert advices before synchronized blocks because the same thread can acquire the lock twice. This behavior can cause a denial of service attack. To clarify more,

```
public class A {
   public void f() {
      // next line is the before advice
      assert !Thread.holdsLock(this);
      synchronized(this){
         /* access files*/
      }
   }
}
```

Figure 3.7: Synchronized Block.

if the thread who owns the lock manipulates files, this will block users from accessing files to which they have access to. A before advice can use Java assertions to check if the lock was hold before entering a synchronized block.

## 3.3 Conclusion

In this chapter, we get two birds in one stone. First, we analyzed the three mostly known AOP approaches from a security point of view, and we retained the pointcut-advice approach as the most appropriate one. Second, we have motivated, from a security point of view, the need for providing new pointcuts in AspectJ. Hence, a description of predicated control flow pointcut and dataflow pointcut and their usefulness from a security point of view are presented. Besides, the importance of loop pointcut to prevent malicious attacks is exposed. A new wildcard for pattern matching is suggested. In addition, we have discussed the need for using modifier pattern like `public` keyword in type pattern syntax. The need for a pointcut to pick out join points associated with setting and getting local variables inside local methods is also discussed as well as new join points for synchronized blocks. In this thesis, we have designed and implemented, as described in Chapter 7, the pointcuts for local variables accesses and the pointcut for the data flow analysis.

# Chapter 4

# JVML Semantics

AspectJ weaving combines the JVML representation (bytecode) of the initial program and the enriched JVML representation of the aspects producing a weaved JVML program representation. As a first step towards the establishment of a semantics for AspectJ, we first establish a semantics for the JVML. The primary objective is to grasp the semantics of Java runtime (Java Virtual Machine Language or JVML) and to compile the underlying meanings into a formal dynamic semantics. JVML is the language interpreted by the Java Virtual Machine (JVM), which is the heart of any Java platform. In this chapter, we will motivate the need for a new JVML semantics, provide the necessary notations and ingredients for the semantics, and finally give the different semantic rules.

## 4.1 Why Another JVML Semantics?

There exist some semantics for JVML, so why we need the definition of a new one? The answer is based on the following arguments:

- In spite of the intensive activities of the research community in formalizing JVML semantics, it remains that there is no contribution that formally addresses, within the same framework, the meanings of JVML features such as multi-threading, synchronization, exception handling, the four method invocations and the use of modifiers.

- Most of the proposed research contributions so far consider only one single thread of execution even though multi-threading is a keystone in Java.

- In the very few proposals where multi-threading has been addressed, it has been done in a way that is not faithful to the official JVML specification. For instance, no distinction is made between implementing the interface `Runnable` or extending the class `Thread`.

Besides these arguments, the main motivation that led us to the formalization of JVML stems from a security investigation of the Java platform. Actually, we needed a formalization that accounts, at the same time and within the same framework, for all the aforementioned JVML aspects. Such a requirement was not satisfied by the state of the art contributions on JVML semantics.

The main traits of the dynamic semantics that we report here are:

- A faithful transcription of the JVML semantics with respect to the official and standard specification [64].

- Thorough and detailed semantic handling of JVML aspects at the same time and within the same framework.

- Handling the semantics of the most tricky features of JVML such as multi-threading, synchronization, exception handling, the four types of method invocation, modifiers, etc.

- A small-step operational style where the evaluation judgements are driven by the syntactic structure of JVML programs.

- A two-layers dynamic semantics: The first layer consists of judgements that capture the semantics of sequential JVML programs in isolation. The second layer consists of judgements that capture the parallel execution of JVML threads.

## 4.2 JVML Semantic Ingredients

In this section, we define the ingredients that we used in the semantic description. Accordingly, we introduce the JVML syntax and we define the notions of data type, computable value, environment, memory store, frame and configuration. A data type refers to a type that is used in JVML. A computable value refers to a dynamic value that is the result of a semantic evaluation of a given JVML expression. An environment is the context that holds the definitions under which the evaluation is done. It corresponds to the current constant pool of a class file. The constant pool is a structure used to represent a class or an interface and JVML instructions refer to the information stored in this structure. Memory store is an abstraction of both the memory storage and the heap. The proposed semantics has the form of a small step operational semantics that is based on evolving configurations. We will use the following notation along this chapter:

- Given two sets $A$ and $B$, $A \underset{m}{\rightharpoonup} B$ denotes the set of all mappings (maps for short) from $A$ to $B$ (partial functions from $A$ to $B$). A map $m \in A \underset{m}{\rightharpoonup} B$ could be defined by extension as $[a_0 \mapsto b_0 \ldots a_{n-1} \mapsto b_{n-1}]$ to denote the association of the elements $b_i$'s to $a_i$'s, where $a_i \in A$ et $b_i \in B$.

- Given a map $m$ from $A$ to $B$, the domain of $m$, $A$, is written Dom($m$).

- Given a partial map $f$, we write $f[x \mapsto v]$ to denote the updating operation of $f$ that yields a map that is equivalent to $f$ except that $x$ is from now on associated with $v$.

- Given a record space $D = \langle f_1 : D_1, f_2 : D_2, \ldots, f_n : D_n \rangle$ and an element $e$ of type $D$, the access to the field $f_i$ of $e$ is written $e.f_i$ and the update of the fields $f_{i1}, \ldots, f_{ik}$ in $e$ by the values $v_{i1}, \ldots, v_{ik} \in D_{i1}, \ldots, D_{ik}$ is written $e[f_{i1} \leftarrow v_{i1}, \ldots, f_{ik} \leftarrow v_{ik}]$. If an update of a field $f_{ij}$ with a value $v_{ij}$ is subjected to a condition $C$, we will use the notation $e.[\ldots, f_{ij} \leftarrow v_{ij}/C, \ldots]$.

- Given a type $\tau$, we write `(`$\tau$`)-list` to denote the type of lists having elements of type $\tau$.

- Given a type $\tau$, we write $(\tau)$ `-set` to denote the type of sets having elements of type $\tau$.
- The space *Identifier* classifies identifiers whereas *NoneType* classifies the unique value `None`, which indicates that there is no specific type.

### 4.2.1 JVML Syntax

Table 4.1 presents a concrete syntax in BNF notation for our JVM set of instructions.

### 4.2.2 Type Algebra

We consider four categories of types: Primitive types, reference types, `void`, and *NoneType*. Reference types are either class types or interface types. An object is a dynamically created class instance and reference values are pointers to these objects. `void` is used to describe the return values of void methods and `None` to describe the return values of constructors. The JVML type algebra is given in Table 4.2.

### 4.2.3 Computable Values

The computable values are presented in Table 4.3. Two kinds of values are considered: Locations and constants. Locations are addresses and constants are values of primitive types. The particular reference value `Null` refers to no object.

### 4.2.4 Environment

We define hereafter the runtime environment and we assume that the reader is familiar with the Java class file format as described in the official specification of JVML [64]. The Java environment as described in Table 4.4 and Table 4.5 models the different declarations in the program and is represented as a map that associates a set of classes to a set of reference types. A class is a record containing a constant pool, a super-class, a set of

*JVMLInstruction* ::= *LocalVariableAccessInstruction*
| *StackManipulationInstruction*
| *ArithmeticInstruction*
| *ConditionalBranchInstruction*
| *UnconditionalBranchInstruction*
| *SynchronizationInstruction*
| *ExceptionInstruction*
| *ObjectAllocationInstruction*
| *MethodCallInstruction*
| *MethodReturnInstruction*
| *FieldAccessInstruction*

*LocalVariableAccessInstruction* ::= `aload` *i* | `iload` *i* | `astore` *i*
| `istore` *i*

*StackManipulationInstruction* ::= `pop` | `push` *n* | `dup`

*ArithmeticInstruction* ::= `iadd`

*ConditionalBranchInstruction* ::= `ifeq` *adr* | `ifne` *adr*

*UnConditionalBranchInstruction* ::= `goto` *adr*

*SynchronizationInstruction* ::= `monitorenter` | `monitorexit`

*ExceptionInstruction* ::= `athrow`

*ObjectAllocationInstruction* ::= `new` *i*

*MethodCallInstruction* ::= `invokevirtual` *i*
| `invokespecial` *i*
| `invokeinterface` *i,n*
| `invokestatic` *i*

*MethodReturnInstruction* ::= `return` | `ireturn` | `areturn`

*FieldAccessInstruction* ::= `getstatic` *i* | `putstatic` *i*
| `getfield` *i* | `putfield` *i*

Table 4.1: JVML Bytecode Grammar

| | | | | |
|---|---|---|---|---|
| *ResultType* | ::= | *Type* | `void` | *NoneType* |
| *Type* | ::= | *PrimitiveType* | *ReferenceType* | |
| *ReferenceType* | ::= | *ClassType* | *InterfaceType* | |
| *ClassType* | ::= | *Identifier* | | |
| *InterfaceType* | ::= | *Identifier* | | |
| *NoneType* | ::= | `None` | | |

Table 4.2: Java Type Algebra

| | | | | |
|---|---|---|---|---|
| *Value* | ::= | *Location* | *Constant* | `Null` |

Table 4.3: Runtime Values

interfaces, a list of fields, a map that associates values to static fields, a list of methods, two flags that indicate whether the class is initialized or not and if the class is an interface, and a monitor. A constant pool is a map that associates a set of integers with a set of constant pool entries. A constant pool entry can be:

- A class type.
- A pair of a method signature and a supposed class.
- A pair of a field signature and a supposed class.

A class type constant pool entry is created, for example, when the compiler encounters a Java instruction `A a = new B()`. The compiler will generate the corresponding `new` and `invokespecial` instructions and a class constant pool entry initialized to `B`. In the two other cases of constant pool entries, the supposed class represents the class in which the method or the field is supposed to be found. We exemplify this with the four following cases where `m` is a method name and `f` a field name:

**Case 1:** Let `o` be an instance class defined in a program `P` as follows: `A o = new B()`.

When encountering a Java expression `o.m()`, respectively `o.f`, in a Java instruction, the compiler will generate a method constant pool entry, respectively a field constant pool

entry, with A as supposed class for those entries.

**Case 2:** Let C be a given class. When encountering a Java expression `C.m()`, respectively `C.f`, in a Java instruction, the compiler will generate a method constant pool entry, respectively a field constant pool entry, with C as supposed class for those entries.

**Case 3:** Let C be a given class. When the compiler encounters a Java expression `m()`, respectively `f`, in a Java instruction inside the class C, the compiler will generate a method constant pool entry, respectively a field constant pool entry, with C as supposed class for those entries.

**Case 4:** Let C be a given class that extends a class D. When the compiler encounters a Java expression `super.m()`, respectively `super.f`, in a Java instruction, the compiler will generate a method constant pool entry, respectively a field constant pool entry, with D as supposed class for those entries.

The value `None` for the super class indicates that the class does not have a super class. The monitor associated with a class is a record of three components: *threadOwner*, *depth* and a *waitList*. If the class is not locked the monitor is set to the value $\langle \texttt{None}, 0, [\,] \rangle$ otherwise the monitor contains the thread identifier that locked the class, the number of times this class has been locked by this same thread and a list of all the threads blocked waiting for this class. A method consists of a method signature, a class name from where the method is, a set of modifiers, a bytecode, a list of method variables and an exception table. A method signature is a record that contains the method's name, the types of arguments, and the result type. The list of the method variables contains the default values of all local variables defined inside the method. The method's parameters are not considered in the method variables. An exception table is a list of exception handlers where an exception handler is defined by:

- Two natural numbers, *startPc* and *endPc*, that are used to determine the code range where the exception handler is valid.
- A natural number, *handler*, that indicates the location that is called upon exception.
- A class type, *exceptionType*, that indicates the class of the exception.

A constructor is considered as a method named `init` with a return type equal to `None` and the class initializer is considered as a static method named `clinit`. A field is represented by a record that contains a field signature, a reference type to which the field belongs to and a set of modifiers. The signature is a combination of the field's name and the field's type.

### 4.2.5 Memory Store

We define, in what follows, a model that captures both the memory storage and the heap of the Java virtual machine. The store as shown in Table 4.6 is a partial mapping from locations to Java objects which are class instances. A Java object is a record containing the class type of the object, a map from the object fields identifiers to run-time values, a monitor and a supplementary information *fromRunnable*. If the object is a "`Thread`" instance constructed from a class that implements the interface "`Runnable`", the name of this class is put in the field *fromRunnable* otherwise *fromRunnable* is set to the value "`None`". This information is useful when a method "`start`" is invoked on an object that is an instance of "`Thread`" or one of its subclasses. It allows to know which "`run`" method to execute. The lookup of "`run`" method starts from the dynamic class of the object if *fromRunnable* is `None` othewrise starts from *fromRunnable*.

An object monitor has the same structure as a class monitor and is set to $\langle \texttt{None}, 0, [\,] \rangle$ if the object is not locked.

| | | |
|---|---|---|
| *JavaEnvironment* | ::= | *ReferenceType* $\underset{m}{\rightarrow}$ *Class* |
| *Class* | ::= | ⟨*constantPool:* *ConstantPool*, |
| | | *superClass:* *RefOrNoneType*, |
| | | *interfaces:* (*InterfaceType*)-`set`, |
| | | *fields:* (*Field*)-`list`, |
| | | *staticMap:* *Field* $\underset{m}{\rightarrow}$ *Value*, |
| | | *methods:* (*Method*)-`list`, |
| | | *initialized:* *Nat*, |
| | | *interface:* *Nat*, |
| | | *monitorClass:* *Monitor* ⟩ |
| *ConstantPool* | ::= | *Nat* $\underset{m}{\rightarrow}$ *ConstantPoolEntry* |
| *ConstantPoolEntry* | ::= | *ClassType* \| *MethodPoolEntry* \| *FieldPoolEntry* |
| *MethodPoolEntry* | ::= | ⟨*methodSignature:* *MethodSignature*, |
| | | *supposedClass:* *ReferenceType* ⟩ |
| *FieldPoolEntry* | ::= | ⟨*fieldSignature:* *FieldSignature*, |
| | | *supposedClass:* *ReferenceType* ⟩ |
| *RefOrNoneType* | ::= | *ReferenceType* \| *NoneType*, |
| *Monitor* | ::= | ⟨*threadOwner:* *ThreadOwner*, |
| | | *depth:* *Nat*, |
| | | *waitList:* *WaitingList* ⟩ |
| *ThreadOwner* | ::= | *ThreadId* \| *NoneType* |
| *WaitingList* | ::= | (*ThreadId*)-`list` |
| *ThreadId* | ::= | *Nat* |
| *Method* | ::= | ⟨*methodSignature:* *MethodSignature*, |
| | | *fromClass:* *ReferenceType*, |
| | | *methodModifiers:* (*MethodModifier*)-`set`, |
| | | *code:* *Code*, |
| | | *methodVariables:* *MethodVariables*, |
| | | *exceptionTable:* *ExceptionTable* ⟩ |

Table 4.4: Java Environment Part I

| | | |
|---|---|---|
| *MethodSignature* | ::= | ⟨name: *Identifier*,<br>*argumentsType:* (*Type*)-list,<br>*resulType:* *ResultType* ⟩ |
| *MethodModifier* | ::= | public \| private \| static \| synchronized |
| *MethodVariables* | ::= | (*Value*)-list |
| *Code* | ::= | *ProgramCounter* $\underset{m}{\rightarrow}$ *JVML Instruction* |
| *ProgramCounter* | ::= | *Nat* |
| *ExceptionTable* | ::= | (*ExceptionHandler*)-list |
| *Field* | ::= | ⟨*fieldSignature:* *FieldSignature*,<br>*fromClass:* *ReferenceType*,<br>*fieldModifiers:* (*FieldModifier*)-set ⟩ |
| *FieldSignature* | ::= | ⟨name: *Identifier*,<br>*type:* *Type* ⟩ |
| *ExceptionHandler* | ::= | ⟨*startPc:* *Nat*,<br>*endPc:* *Nat*,<br>*handler:* *Nat*,<br>*exceptionType:* *ClassType* ⟩ |
| *FieldModifier* | ::= | public \| private \| static |

Table 4.5: Java Environment Part II

| | | | |
|---|---|---|---|
| *Store* | ::= | *Location* $\underset{m}{\rightarrow}$ *JavaObject* | |
| *JavaObject* | ::= | ⟨*classType:* | *ClassType*, |
| | | *fieldsMap:* | *Field* $\underset{m}{\rightarrow}$ *Value*, |
| | | *monitor:* | *Monitor* |
| | | *fromRunnable:* | *ClassType* \| *NoneType* ⟩ |
| | | *depth:* | *Nat*, |
| | | *waitList:* | *WaitingList* ⟩ |

Table 4.6: Store Structure

### 4.2.6 Frame

A frame is a runtime data structure that captures the execution state of a JVML method. It is defined as a tuple $\{|m, pc, l, o, z|\}$ where:

- $m$ is the current method.
- $pc$ represents the program counter that contains the address of the instruction to be executed in the method $m$.
- $l$ contains the values of the different local variables of $m$.
- $o$ represents a stack of operand values.
- $z$ is the element locked in case where $m$ is a synchronized method. It is the class locked in the case where the method is static or the reference to the object locked in the case where the method is not static. The value is `None` in case of non-synchronized methods.

A formal description for the frames is given in Table 4.7.

### 4.2.7 Configurations

The operational semantics is based on the evolution of configurations that are defined hereafter. As stated previously, our dynamic semantics uses two layers. The first layer captures the semantics of sequential programs whereas the second layer is meant to capture the semantics of multi-threading. Therefore, we need to introduce two categories of configurations. The domain of configurations that are dedicated to mono-threaded programs is *ThreadConfiguration*. The domain of configurations that are dedicated to multi-threaded programs is *MultiThreadConfiguration*. These two categories are respectively defined in Table 4.8 and Table 4.9. A thread configuration will have the following form:

| | | | |
|---|---|---|---|
| *MethodFrame* | ::= | ⟨ *method:* | *Method,* |
| | | *programCounter:* | *ProgramCounter* , |
| | | *locals:* | *Locals,* |
| | | *operandStack:* | *OperandStack,* |
| | | *synchronizedElement:* | *SynchronizedElement* ⟩ |
| *Locals* | ::= | (*Value*) `-list` | |
| *OperandStack* | ::= | (*Value*) `-list` | |
| *SynchronizedElement* | ::= | *ClassOrLocation* \| *NoneType* | |
| *ClassOrLocation* | ::= | *Location* \| *ReferenceType* | |

Table 4.7: Method Frame

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, x \rangle$$

where:

- $\mathcal{JE}$ represents the environment.
- $\mathcal{S}$ is the store.
- $\{|m, pc, l, o, z|\} :: \mathcal{F}$ represents the thread's stack from which method frames are retrieved. The term $\{|m, pc, l, o, z|\}$ denotes the frame that is the top element of frame stack.
- $\mathcal{L}$ contains the objects and classes that are locked by the current thread.
- $\iota$ represents the identity of the current thread, i.e. the one executing the method $m$.
- $x$ indicates if an exception has been detected. Whenever an exception is raised, we will use the location $e$ to point to the underlying object that is an instance of the class `Throwable`. If no exception is thrown `None` is used instead.

For each thread, we maintain a list of objects and classes that have been locked by this thread. Exceptions can be thrown explicitly using the `athrow` instruction or

| | | | |
|---|---|---|---|
| *ThreadConfiguration* | ::= | *JavaEnvironment*×*Store*×*ThreadInformation* | |
| *ThreadInformation* | ::= | ⟨*threadStack:* | *ThreadStack*, |
| | | *lockedElements:* | *LockedElements* |
| | | *threadId:* | *ThreadId* |
| | | *exception:* | *Exception* ⟩ |
| *ThreadStack* | ::= | (*MethodFrame*)-`list` | |
| *LockedElements* | ::= | (*ClassOrLocation*)-`list` | |
| *Exception* | ::= | *Location* \| *NoneType* | |

Table 4.8: Thread Configurations

| | | | |
|---|---|---|---|
| *MultiThreadConfiguration* | ::= | *JavaEnvironment*×*Store*×*JavaStack* | |
| *JavaStack* | ::= | *ThreadId* $\underset{m}{\rightharpoonup}$ *Thread* | |
| *Thread* | ::= | ⟨*threadInformation:* | *ThreadInformation*, |
| | | *state:* | *State* ⟩ |
| *State* | ::= | `active` \| `blocked` | |

Table 4.9: Multi-Threads Configurations

implicitly by the virtual machine when runtime tests fail as passing a null pointer to a `getfield` instruction. In case of an exception, *e* points to the respective exception object in the store.

The configuration in Table 4.9 is used in presence of multiple threads and is a combination of an environment, a store, and a Java stack. The Java stack contains information about the current threads and consists of a partial mapping that associates thread information and state to the *Nat* number that identifies the thread. The term `blocked` is used to denote the state of a thread waiting for a resource, owned by another thread, otherwise the thread is said to be `active`.

## 4.3 JVML Semantic Rules

This section presents the JVML semantics rules. This semantics is structured in two layers, one for threads in isolation and another for threads running in parallel following [47]. For processes in isolation, the semantics is defined using a labelled transition system on thread configurations i.e. (*ThreadConfiguration*, $\Lambda$, $\longrightarrow$) whereas an unlabelled state transition system (*MultiThreadConfiguration*, $\hookrightarrow$) is used for multisets of threads. The set of labels $\Lambda$ is defined as follows:

$$
\begin{array}{lll}
\Lambda \ni \ell & ::= & \varepsilon \\
& | & \mathit{block} \ldots\ldots\ldots\ldots\ldots\ldots \mathit{Block\ Current\ Thread} \\
& | & \mathit{kill} \ldots\ldots\ldots\ldots\ldots\ldots\ldots \mathit{Kill\ Current\ Thread} \\
& | & \mathit{run}(\mathit{class} : \mathit{ClassType}) \ldots\ldots\ldots\ldots \mathit{Fork\ New\ Thread} \\
& | & \mathit{notify}(x : \mathit{ClassOrLocation}) \ldots \mathit{Notify\ Blocked\ Threads}
\end{array}
$$

The labels on the transitions contain the information to send from the first layer to the second one. The transition label $\varepsilon$[1] allows to report, in the Java stack, all the modifications that the current thread has been subjected to during this transition. The transition label *block* is used to change the current thread's state from `active` to `blocked`. The *kill* label refers to the case where the current thread must be killed and thus its corresponding entry in the Java stack should be removed. The label *run*(*class*:*ClassType*) reports to the second layer that a new thread must be created and that the lookup of its `run` method must start from the parameter *class*. The last transition label *notify*(*x*:*ClassOrLocation*) is used by the second layer in order to change the state of all the threads waiting for the resource $x$ from `blocked` to `active`.

[1]For the rest of the paper, we adopt the notation $C_1 \longrightarrow C_2$ instead of $C_1 \xrightarrow{\varepsilon} C_2$.

### 4.3.1 First Layer

In the following, we present the different semantic rules when considering solely one thread.

#### Local Variable Access

The local variable access instructions rules consist either in loading variables or addresses from the local variables list of a method to its operand stack or in storing[et values or addresses from the operand stack into the locals variables list. When one of those rules processes, it increments the program counter of the current thread and updates the operand stack by popping or pushing one element. The semantic rules related to the local variable access instructions are the following:

$$\frac{\begin{array}{c} m.code(pc) = \texttt{aload}\ i \\ o' = \mathsf{getLocalValue}(l,i) :: o \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{iload}\ i \\ o' = \mathsf{getLocalValue}(l,i) :: o \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{astore}\ i \\ Loc = \mathsf{getOneStackElem}(o,0) \\ l' = \mathsf{setLocalValue}(l,i,Loc) \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l', o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{istore}\ i \\ v = \mathsf{getOneStackElem}(o,0) \\ l' = \mathsf{setLocalValue}(l,i,v) \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,pc+1,l',o',z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

### Stack Manipulation

The following instructions manipulate the stack of operands by popping an element from the stack, pushing an element in the stack, or duplicating the top of the stack. After running one of those rules, the program counter is incremented and the operand stack updated in the new current thread configuration. The semantic rules related to the stack manipulation instructions are the following:

$$\frac{\begin{array}{c} m.code(pc) = \texttt{pop} \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,pc+1,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{push}\ n \\ o' = \mathsf{pushStack}(o,n) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,pc+1,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{dup} \\ o' = \mathsf{pushStack}(o,\mathsf{head}(o)) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,pc+1,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

### Arithmetic Operation

The rule of the bytecode `iadd` consists of adding the two values on the top of the stack, popping them and pushing the result instead. When the rule is fired, the operand stack and the program counter are updated in the new current thread configuration.

$$\frac{m.code(pc) = \texttt{iadd} \qquad o' = (\mathsf{getOneStackElem}(o,0) + \mathsf{getOneStackElem}(o,1)) :: \mathsf{popStack}(o,2)}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,pc+1,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle}$$

**Branch Statements**

The conditional branch instruction `ifeq` *adr*, respectively `ifne` *adr*, tests the two values on the top of the operand stack and performs a branching to the address *adr* when the two values are equal, respectively different. The unconditional branch instruction `goto` *adr* changes, without any condition, the program's counter of the current method to the address specified in the bytecode. When a branch instruction rule is fired, the program counter in the new thread configuration is set to *adr*. In the case of a conditional branch instruction rule, the operand stack is also updated (by popping the two elements on the top). The semantic rules related to the branch instructions are the following:

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ifeq}\ adr \\ \mathsf{getOneStackElem}(o,0) = \mathsf{getOneStackElem}(o,1) \\ o' = \mathsf{popStack}(o,2) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,adr,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ifne}\ adr \\ \mathsf{getOneStackElem}(o,0) \neq \mathsf{getOneStackElem}(o,1) \\ o' = \mathsf{popStack}(o,2) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,adr,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle}$$

$$\frac{m.code(pc) = \texttt{goto}\ adr}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m,adr,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \mathtt{None}\rangle}$$

**Synchronization Statement:** `monitorenter`

The thread that executes `monitorenter` tries to gain ownership of the monitor associated with the object reference on the top of the stack whereas it exits it when executing `monitorexit`.

The first rule corresponds to the case where the reference on the top of the operand stack is not `Null` and the referenced object is not owned by another thread. In this case, the current thread becomes (or stays) the owner of the object and increments the number of times it has entered the corresponding monitor. When this rule processes, the store, the program counter, the operand stack, and in some cases the list of elements locked by the current thread are updated . In fact, the monitor information of the gained object is updated by the store, the reference of the acquired object is popped from the operand stack, and the program counter is incremented. If the object acquired was not already owned by the current thread, it will be added to the list of its locked elements.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorenter} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ (\neg\ \mathsf{isLocked}(\mathcal{S},Loc) \vee \mathsf{isOwner}(\mathcal{S},Loc,\iota)) \wedge Loc \neq \texttt{Null} \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorEntered}(\mathcal{S},Loc,\iota)] \\ \mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isOwner}(\mathcal{S},Loc,\iota),\mathcal{L},Loc :: \mathcal{L}) \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S}',\{|m,pc+1,l,o',z|\} :: \mathcal{F},\mathcal{L}',\iota,\texttt{None}\rangle}$$

The second rule refers to the case where the reference on the top of the operand stack is not `Null` but the referenced object is owned by another thread. In this case, the current thread is added to the waiting list of the considered object implying an update of the store. The transition label *block* is used to mean that the current thread must be blocked.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorenter} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ \mathsf{isLocked}(\mathcal{S},Loc) \wedge \neg\ \mathsf{isOwner}(\mathcal{S},Loc,\iota) \wedge Loc \neq \texttt{Null} \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S},Loc,\iota)] \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \xrightarrow{block} \langle \mathcal{JE},\mathcal{S}',\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle}$$

The third rule depicts the case where the reference on the top of the stack, on which the `monitorenter` is invoked, is `Null`. In this case, a `Throwable` object is created

in the store and its reference is assigned to the thread configuration exception flag.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorenter} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ Loc = \texttt{Null} \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable}]\ ;\ e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

**Synchronization Statement:** `monitorexit`

Three rules describe the semantics of `monitorexit`. The first rule corresponds to the case where the object reference on the top of the operand stack is not `Null` and the corresponding object is owned by the current thread. In this case, the thread decrements the counter indicating the number of times the thread has entered this monitor. This rule describes the case where after decrementing this counter, it remains greater than 0. Consequently, the current thread will not release the monitor. When this rule processes, the monitor information of the considered object is updated by the store, the reference of the object is popped from the operand stack, and the program counter is incremented.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorexit} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ Loc \neq \texttt{Null} \wedge \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, Loc, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', Loc) \neq 0 \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m,pc,l,o,z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m,pc+1,l,o',z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The second rule refers to the same case as the previous rule except that in this case the thread releases the object. As in the first rule, the thread decrements the counter indicating the number of times the thread has entered this monitor. The difference here is that after decrementing this counter, its value will be equal to 0. When this rule processes, the monitor information of the released object is updated by the store, the reference of the

released object is popped from the operand stack, and the program counter is incremented. Furthermore, the reference of the released object is suppressed from the list of the current thread locked elements and the transition label *notify(Loc)* is used to notify after all the threads waiting for the object referenced by *Loc*.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorexit} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ Loc \neq \texttt{Null} \wedge \mathsf{isOwner}(\mathcal{S},Loc,\iota) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorExited}(\mathcal{S},Loc,\iota)] \\ \mathsf{depthLock}(\mathcal{S}',Loc) = 0 \\ \mathcal{L}' = \mathsf{suppress}(Loc,\mathcal{L}) \\ o' = \mathsf{popStack}(o,1) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \xrightarrow{notify(Loc)} \langle \mathcal{JE},\mathcal{S}',\{|m,pc+1,l,o',z|\} :: \mathcal{F},\mathcal{L}',\iota,\texttt{None}\rangle}$$

The third rule depicts the case where either the object reference on the top of the stack, on which the `monitorexit` is invoked, is `Null` or the monitor is not owned by the current thread. A `Throwable` object is created in the store and its reference is assigned to the thread configuration exception flag.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{monitorexit} \\ \neg\ \mathsf{isOwner}(\mathcal{S},Loc,\iota)\ \vee Loc = \texttt{Null} \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE},\texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S}',\{|m,pc,l,Loc :: o,z|\} :: \mathcal{F},\mathcal{L},\iota,e\rangle}$$

**Exception Handling**

We consider two rules for the `athrow` instruction depending on whether the execution of `athrow` results in runtime exceptions or not. In the JVM specification [64] runtime exceptions result in the instruction `athrow` either when the object reference on the top of the stack is null or when the method of the current frame is a synchronized method and the current thread is not the owner of the monitor acquired or reentered on invocation of this method.

The following rule describes the case without runtime exception. The current configuration moves to a configuration where the exception flag is set to the object reference on the top of the stack.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{athrow} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ (Loc \neq \texttt{Null}) \wedge (\neg \mathsf{isSynchronized}(m) \vee \mathsf{isOwner}(\mathcal{S},z,\iota)) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,Loc\rangle}$$

The second rule describes the case where a runtime exception arises. In the new thread configuration, a new `Throwable` object is added to the store and an exception flag is set to the reference of this new `Throwable` object.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{athrow} \\ Loc = \mathsf{getOneStackElem}(o,0) \\ (Loc = \texttt{Null}) \vee (\mathsf{isSynchronized}(m) \wedge \neg \mathsf{isOwner}(\mathcal{S},z,\iota)) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE},\texttt{Throwable})] \; ; \; e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S}',\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,e\rangle}$$

Three other semantic rules specify how the semantic configurations are handled in case where the exception flag is different from `None`. In this case, an exception is thrown and we must check if the calling method is prepared to catch the exception.

The first rule describes the case where the calling method is prepared to catch the exception and contains an appropriate handler for the exception referenced by the flag $e$ (Return value of appropriatePcHandler is different from $-1$) . In the new thread configuration, the program counter of the current method is set to the value indicated by the handler, the operand stack is cleared, $e$ is pushed back onto the operand stack and the exception flag is set to `None`.

$$\frac{\begin{array}{c} type = \mathsf{getDynamicClass}(\mathcal{S},e) \\ pcH = \mathsf{appropriatePcHandler}(\mathcal{JE},pc,type,m.exceptionTable) \\ pcH \neq -1 \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,e\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S},\{|m,pcH,l,e,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle}$$

In the second rule, the method on the top of the frame stack does not contain an appropriate handler. In the new thread configuration, the frame is popped allowing the search of an appropriate handler from the invoker.

$$\frac{\begin{array}{c} type = \mathsf{getDynamicClass}(\mathcal{S}, e) \\ \mathsf{appropriatePcHandler}(\mathcal{JE}, pc, type, m.exceptionTable) = -1 \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \{|n, pc', l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, e\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|n, pc', l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, e\rangle}$$

The third rule describes the case where the exception flag in the initial thread configuration is an exception reference $e$ and where the frame stack contains only one method. Furthermore, this unique method in the frame stack cannot handle the exception referenced by $e$. In this case, the information that the current thread must be killed is sent to the second layer using the transition label *kill* and nothing is changed in the configuration.

$$\frac{\begin{array}{c} type = \mathsf{getDynamicClass}(\mathcal{S}, e) \\ \mathsf{appropriatePcHandler}(\mathcal{JE}, pc, type, m.exceptionTable) = -1 \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\}, \mathcal{L}, \iota, e\rangle \xrightarrow{kill} \langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\}, \mathcal{L}, \iota, e\rangle}$$

### Object Creation

Four rules describe the semantics of the New instruction, which is used to create new objects. The first rule refers to the case where the class of the object to create is not an interface and is initialized. When the rule fires, a new object is then created in the store, the corresponding reference is pushed onto the operand stack, and the program counter is incremented.

$$\frac{\begin{array}{c} m.code(pc) = \mathtt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i) \\ \neg\, \mathsf{isInterface}(\mathcal{JE}, ct) \wedge \mathsf{isInitialized}(\mathcal{JE}, ct) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{newObject}(\mathcal{JE}, ct)]\ ;\ Loc \notin Dom(\mathcal{S}) \\ o' = \mathsf{pushStack}(o, Loc) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \mathtt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \mathtt{None}\rangle}$$

The second rule describes the case where the class of the object to create is not an interface, is not initialized, and is not locked by another thread. In this case, a frame of

its `clinit`[2] method is pushed onto the frame stack. Then, the class is locked and added to the locked elements of the thread if it is not there yet. In addition, the monitor of the considered class is updated to reflect the fact that the current thread has acquired it or reentred it. This implies the update of the environment.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i) \\ \neg\ \mathsf{isInterface}(\mathcal{JE}, ct) \wedge \neg\ \mathsf{isInitialized}(\mathcal{JE}, ct) \\ \neg\ \mathsf{isClassLocked}(\mathcal{JE}, ct) \vee \mathsf{isClassOwner}(\mathcal{JE}, ct, \iota) \\ signatureClinit = \langle \texttt{clinit}, [\,], \texttt{void} \rangle \\ clinit = \mathsf{retrieveM}(signatureClinit, \mathcal{JE}(ct).methods) \\ clinitFrame = \mathsf{newFrame}(clinit, 0, clinit.methodVariables, [\,], ct) \\ \mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{JE}, ct, \iota), \mathcal{L}, ct :: \mathcal{L}) \\ \mathcal{JE}' = \mathcal{JE}[ct \mapsto \mathsf{classMonitorEntered}(\mathcal{JE}, ct, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, clinitFrame :: \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}$$

The third rule reflects the case where the class of the object to be created is not an interface, is not initialized but is locked by another thread. In this case, the thread identity is put in the waiting list of the class implying an update of the environment. Furthermore, a signal is sent the second layer with the label *block* in order to block the current thread.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i) \\ \neg\ \mathsf{isInterface}(\mathcal{JE}, ct) \wedge \neg\ \mathsf{isInitialized}(\mathcal{JE}, ct) \\ \mathsf{isClassLocked}(\mathcal{JE}, ct) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{JE}, ct, \iota) \\ \mathcal{JE}' = \mathcal{JE}[ct \mapsto \mathsf{addToClassWaitingList}(\mathcal{JE}, ct, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{block} \langle \mathcal{JE}', \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The fourth rule represents the case where the class of the object to create is an interface. A `Throwable` object is created in the store and its reference is assigned to the thread configuration exception flag.

[2]We consider that each class has a `clinit` method even it is empty.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{new}\ i \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i) \\ \mathsf{isInterface}(\mathcal{JE}, ct) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

**Method Invocation:** `invokevirtual`

The `invokevirtual` instruction invokes instance methods and its semantics is described with the following five rules. According to the specification [64], the method lookup for the `invokevirtual` instruction starts always from the dynamic class of the reference object on the top of the operand stack.

The first rule corresponds to the case where the invoked method is resolved, found, not synchronized, and is not the `start` method of the class `Thread` or one of its subclasses. The arguments values and the object reference are then popped from the current operand stack and a new frame is created for the method being invoked. The object reference and the argument values become the values of local variables of the new frame. We add to local variables the internal variables of the method taken from the environment.

$$
\frac{\begin{array}{c}
m.code(pc) = \texttt{invokevirtual}\ i \\
ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\
ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\
\mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\
argCount = \mathsf{length}(ms.argumentsTypes) \\
Loc = \mathsf{getOneStackElem}(o, argCount) \\
Loc \neq \texttt{Null} \\
dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\
m' = \mathsf{lookupM}(\mathcal{JE}, ms, dc) \\
m' \neq \texttt{None} \wedge (m'.name \neq \texttt{start} \vee \neg \mathsf{isThread}(m'.fromClass)) \\
\neg\, \mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \neg\, \mathsf{isStaticM}(m') \\
l' = m'.methodVariables \\
l'' = \mathsf{getStackElemts}(o, argCount) \\
o' = \mathsf{popStack}(o, argCount + 1) \\
nf = \mathsf{newFrame}(m', 0, Loc :: l'' :: l', [\,], \texttt{None})
\end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}
$$

The second rule describes the case of `invokevirtual` when the invoked method is resolved and the result of the lookup is the `start` method of the class `Thread` or one of its subclasses. In this case, an information is sent to the second level using the label *run(class)* in order to start running the new thread. The parameter *class* of the label represents the class from which the lookup for the method `run` of the new thread starts. If the object referenced by the top of the operand stack was constructed using a `Runnable` interface then the field *fromRunnable* of this object is assigned to the variable *class*. If the object referenced by the top of the operand stack was not constructed using a `Runnable` interface (i.e. the field *fromRunnable* is `None`) then its dynamic class (i.e. the class `Thread` or one of its subclasses from which the object has been instantiated) is assigned to the variable *class*. When this rule processes, the reference on the top of the operand stack of the current thread is popped and the current thread program counter is incremented. The reader is invited to read the first rule of `invokespecial` to better understand the use of the field *fromRunnable* of store's objects.

$$
\frac{\begin{array}{c} m.code(pc) = \texttt{invokevirtual}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ Loc = \mathsf{getOneStackElem}(o, argCount) \\ Loc \neq \texttt{Null} \\ dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\ m' = \mathsf{lookupM}(\mathcal{JE}, ms, dc) \\ m' \neq \texttt{None} \\ m'.name = \texttt{start} \wedge \mathsf{isThread}(m'.fromClass) \\ class = \mathsf{ifThenElse}(\mathcal{S}(Loc).fromRunnable = \texttt{None}, dc, \mathcal{S}(Loc).fromRunnable) \\ o' = \mathsf{popStack}(o, 1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle \xrightarrow{run(class)} \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle}
$$

The third case is as the first one except that the method is synchronized and that the monitor associated with the receiver can be acquired or reentered and is not owned by another thread. Contrarily to the first rule, the store updates the monitor information of the gained object and if this latter was not already owned by the current thread, it will be added to the list of its locked elements. Furthermore, the field *synchronizedElement* of the new frame is set to the top of the operand stack of the current method whereas it was `None` in the first rule of `invokevirtual`.

$$
\frac{\begin{array}{c}
m.code(pc) = \texttt{invokevirtual}\ i \\
ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\
ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\
\mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\
argCount = \mathsf{length}(ms.argumentsTypes) \\
Loc = \mathsf{getOneStackElem}(o, argCount) \\
Loc \neq \texttt{Null} \\
dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\
m' = \mathsf{lookupM}(\mathcal{JE}, ms, dc) \\
m' \neq \texttt{None} \\
\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \neg\, \mathsf{isStaticM}(m') \\
\neg\, \mathsf{isLocked}(\mathcal{S}, Loc) \vee \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \\
\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isOwner}(\mathcal{S}, Loc, \iota), \mathcal{L}, Loc :: \mathcal{L}) \\
\mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorEntered}(\mathcal{S}, Loc, \iota)] \\
l' = m'.methodVariables \\
l'' = \mathsf{getStackElemts}(o, argCount) \\
o' = \mathsf{popStack}(o, argCount + 1) \\
nf = \mathsf{newFrame}(m', 0, Loc :: l'' :: l', [\,], Loc)
\end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}
$$

The fourth case is similar to the previous one except that the monitor associated to the receiver is already owned by another thread. The current thread is added to the waiting list of the receiver (i.e. the object referenced by the top of the current thread's operand stack) and the store is updated. The transition label *block* transmits to the second layer the information that the current thread must be blocked.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokevirtual}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ Loc = \mathsf{getOneStackElem}(o, argCount) \\ Loc \neq \texttt{Null} \\ dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\ m' = \mathsf{lookupM}(\mathcal{JE}, ms, dc) \\ m' \neq \texttt{None} \\ \mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \neg\, \mathsf{isStaticM}(m') \\ \mathsf{isLocked}(\mathcal{S}, Loc) \wedge \neg\, \mathsf{isOwner}(\mathcal{S}, Loc, \iota) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S}, Loc, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{block} \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The fifth and last rule corresponds to the case where an exception occurs. This happens when the method is not resolved or when the lookup fails or when the method is a static one. It happens also when the access to this invoked method is not allowed or finally when the receiver on the stack is `Null`. In this case, a `Throwable` object is created in the store and its reference is assigned to the thread configuration exception flag.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokevirtual}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ Loc = \mathsf{getOneStackElem}(o, argCount) \\ dc = \mathsf{getDynamicClass}(\mathcal{S}, Loc) \\ m' = \mathsf{lookupM}(\mathcal{JE}, ms, dc) \\ C1 = (Loc = \texttt{Null}) \vee (\ m' = \texttt{None}) \\ C1 \vee \neg\ \mathsf{accessAllowedM}(m, m') \vee\ \mathsf{isStaticM}(m') \vee \neg \mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

**Method Invocation:** `invokeinterface`

The bytecode `invokeinterface` is used to invoke an interface method and we use four rules to describe its semantics. These rules are very similar to the first, third, fourth and fifth rule of `invokevirtual` except that the resolution uses the function isMethInterfaceResolved instead of isMethResolved and that the method must be a public method. We have then the four following rules:

$$m.code(pc) = \texttt{invokeinterface}\ i,n$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).supposedClass$$
$$\mathsf{isMethInterfaceResolved}(\mathcal{JE},ms,ct)$$
$$Loc = \mathsf{getOneStackElem}(o,n)$$
$$Loc \neq \texttt{Null}$$
$$dc = \mathsf{getDynamicClass}(\mathcal{S},Loc)$$
$$m' = \mathsf{lookupM}(\mathcal{JE},ms,dc)$$
$$m' \neq \texttt{None}$$
$$\neg\,\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m,m') \wedge \neg\,\mathsf{isStaticM}(m') \wedge \mathsf{isPublicM}(m')$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getAllStackElemts}(o,n)$$
$$o' = \mathsf{popStack}(o,n+1)$$
$$nf = \mathsf{newFrame}(m',0,Loc :: l'' :: l',[\,],\texttt{None})$$

---

$$\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\mathfrak{l},\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S},nf :: \{|m,pc,l,o',z|\} :: \mathcal{F},\mathcal{L},\mathfrak{l},\texttt{None}\rangle$$

$$m.code(pc) = \texttt{invokeinterface}\ i,n$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).supposedClass$$
$$\mathsf{isMethInterfaceResolved}(\mathcal{JE},ms,ct)$$
$$Loc = \mathsf{getOneStackElem}(o,n)$$
$$Loc \neq \texttt{Null}$$
$$dc = \mathsf{getDynamicClass}(\mathcal{S},Loc)$$
$$m' = \mathsf{lookupM}(\mathcal{JE},ms,dc)$$
$$m' \neq \texttt{None}$$
$$\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m,m') \wedge \neg\, \mathsf{isStaticM}(m') \wedge \mathsf{isPublicM}(m')$$
$$\neg\, \mathsf{isLocked}(\mathcal{S},Loc) \vee \mathsf{isOwner}(\mathcal{S},Loc,\iota)$$
$$\mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorEntered}(\mathcal{S},Loc,\iota)]$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getAllStackElemts}(o,n)$$
$$o' = \mathsf{popStack}(o,n+1)$$
$$nf = \mathsf{newFrame}(m',0,Loc :: l'' :: l',[\,],Loc)$$
$$\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isOwner}(\mathcal{S},Loc,\iota),\mathcal{L},Loc :: \mathcal{L})$$

---

$$\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\iota,\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S}',nf :: \{|m,pc,l,o',z|\} :: \mathcal{F},\mathcal{L}',\iota,\texttt{None}\rangle$$

$$
\frac{\begin{array}{c} m.code(pc) = \texttt{invokeinterface}\ i,n \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).supposedClass \\ \mathsf{isMethInterfaceResolved}(\mathcal{JE},ms,ct) \\ Loc = \mathsf{getOneStackElem}(o,n) \\ Loc \neq \texttt{Null} \\ dc = \mathsf{getDynamicClass}(\mathcal{S},Loc) \\ m' = \mathsf{lookupM}(\mathcal{JE},ms,dc) \\ m' \neq \texttt{None} \\ \mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m,m') \wedge \neg\, \mathsf{isStaticM}(m') \wedge \mathsf{isPublicM}(m') \\ \mathsf{isLocked}(\mathcal{S},Loc) \wedge \neg\, \mathsf{isOwner}(\mathcal{S},Loc,\mathfrak{t}) \\ \mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S},Loc,\mathfrak{t})] \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\mathfrak{t},\texttt{None}\rangle \xrightarrow{block} \langle \mathcal{JE},\mathcal{S}',\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\mathfrak{t},\texttt{None}\rangle}
$$

$$
\frac{\begin{array}{c} m.code(pc) = \texttt{invokeinterface}\ i,n \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).methodSignature \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE},m,i).supposedClass \\ Loc = \mathsf{getOneStackElem}(o,n) \\ dc = \mathsf{getDynamicClass}(\mathcal{S},Loc) \\ m' = \mathsf{lookupM}(\mathcal{JE},ms,dc) \\ C = (Loc = \texttt{Null} \vee m' = \texttt{None} \vee \neg\, \mathsf{accessAllowedM}(m,m')) \\ C \vee \mathsf{isStaticM}(m') \vee \neg \mathsf{isPublicM}(m') \neg \mathsf{isMethInterfaceResolved}(\mathcal{JE},ms,ct) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE},\texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE},\mathcal{S},\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\mathfrak{t},\texttt{None}\rangle \longrightarrow \langle \mathcal{JE},\mathcal{S}',\{|m,pc,l,o,z|\} :: \mathcal{F},\mathcal{L},\mathfrak{t},e\rangle}
$$

**Method Invocation:** `invokestatic`

Contrarily to `invokevirtual` and `invokeinterface`, `invokestatic` is not used to invoke an instance method but a class method resulting in the absence of an instance reference in the operand stack of the method. Another difference is that if the method is synchronized, we must acquire or reenter the monitor associated with the class where the method is defined (and not the object on which the call is done). As in the

cases of `invokevirtual` or `invokeinterface`, when the monitor is owned by another thread the current thread is blocked. We have six rules describing the semantics of `invokestatic`, the four first ones are very close to the four rules of `invokeinterface` whereas the last two rules represent the case where the invoked method's class is not initialized. The fifth rule represents the case where the frame of `clinit` that initializes the class can be pushed onto the frame stack. The sixth rule refers to the case where the monitor of the non-initialized class is owned by another thread resulting in blocking the current thread.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokestatic}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\ m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct) \\ m' \neq \texttt{None} \\ cm = m'.fromClass \\ \neg\, \mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \mathsf{isStaticM}(m') \wedge \mathsf{isInitialized}(\mathcal{JE}, cm) \\ l' = m'.methodVariables \\ l'' = \mathsf{getStackElemts}(o, argCount) \\ o' = \mathsf{popStack}(o, argCount) \\ nf = \mathsf{newFrame}(m', 0, l'' :: l', [\,], \texttt{None}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle}$$

$$m.code(pc) = \texttt{invokestatic}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct)$$
$$m' \neq \texttt{None}$$
$$cm = m'.fromClass$$
$$\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \mathsf{isStaticM}(m') \wedge \mathsf{isInitialized}(\mathcal{JE}, cm)$$
$$\neg\, \mathsf{isClassLocked}(\mathcal{JE}, cm) \vee \mathsf{isClassOwner}(\mathcal{JE}, cm, \iota)$$
$$\mathcal{JE}' = \mathcal{JE}[cm \mapsto \mathsf{classMonitorEntered}(\mathcal{JE}, cm, \iota)]$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getStackElemts}(o, argCount)$$
$$o' = \mathsf{popStack}(o, argCount)$$
$$nf = \mathsf{newFrame}(m', 0, l'' :: l', [\,], cm)$$
$$\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{JE}, cm, \iota), \mathcal{L}, cm :: \mathcal{L})$$

---

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle$$

$$m.code(pc) = \texttt{invokestatic}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct)$$
$$m' \neq \texttt{None}$$
$$cm = m'.fromClass$$
$$\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \mathsf{isStaticM}(m') \wedge \mathsf{isInitialized}(\mathcal{JE}, cm)$$
$$\mathsf{isClassLocked}(\mathcal{JE}, cm) \wedge \neg\, \mathsf{isClassOwner}(\mathcal{JE}, cm, \iota)$$
$$\mathcal{JE}' = \mathcal{JE}[cm \mapsto \mathsf{addToClassWaitingList}(\mathcal{JE}, cm, \iota)]$$

---

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{block} \langle \mathcal{JE}', \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle$$

$$m.code(pc) = \texttt{invokestatic}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct)$$
$$m' = \texttt{None} \vee \neg\, \mathsf{accessAllowedM}(\texttt{m}, \texttt{m}') \vee \neg\, \mathsf{isStaticM}(\texttt{m}') \vee \neg \mathsf{isMethResolved}(\mathcal{JE}, \texttt{ms}, \texttt{ct})$$
$$\mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S})$$
$$\overline{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e\rangle}$$

$$m.code(pc) = \texttt{invokestatic}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct)$$
$$cm = m'.fromClass$$
$$\neg\, \mathsf{isInitialized}(\mathcal{JE}, cm)$$
$$\neg\, \mathsf{isClassLocked}(\mathcal{JE}, cm) \vee \mathsf{isClassOwner}(\mathcal{JE}, cm, \iota)$$
$$signatureClinit = \langle \texttt{clinit}, [\,], \texttt{void}\rangle$$
$$clinit = \mathsf{retrieveM}(signatureClinit, \mathcal{JE}(cm).methods)$$
$$clinitFrame = \mathsf{newFrame}(clinit, 0, clinit.methodVariables, [\,], cm)$$
$$\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{JE}, cm, \iota), \mathcal{L}, cm :: \mathcal{L})$$
$$\mathcal{JE}' = \mathsf{ifThenElse}(\mathsf{isSynchronized}(m), \mathcal{JE}[cm \mapsto \mathsf{classMonitorEntered}(\mathcal{JE}, cm, \iota)], \mathcal{JE})$$
$$\overline{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, clinitFrame :: \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{invokestatic}\ i \\ ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature \\ argCount = \mathsf{length}(ms.argumentsTypes) \\ ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ \mathsf{isMethResolved}(\mathcal{JE}, ms, ct) \\ m' = \mathsf{lookupM}(\mathcal{JE}, ms, ct) \\ cm = m'.fromClass \\ \neg\ \mathsf{isInitialized}(\mathcal{JE}, cm) \\ \mathsf{isClassLocked}(\mathcal{JE}, cm) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{JE}, cm, \iota) \\ \mathcal{JE}' = \mathcal{JE}[cm \mapsto \mathsf{addToClassWaitingList}(\mathcal{JE}, cm, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{block} \langle \mathcal{JE}', \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

**Method Invocation:** `invokespecial`

Five rules describe the semantics of the instruction `invokespecial`. The first rule corresponds to the case where the method is `init` and there is no exceptions. In this case, we retrieve the method inside the current class. Furthermore, if the class where the method `init` is supposed to be found is the `Thread` class and if the first argument of the method is a class $x$ that implements either `Runnable` itself or one of its subclasses, we must put $x$ in the field *fromRunnable* of the thread object. This information is needed in the second rule of `invokevirtual`.

$$m.code(pc) = \texttt{invokespecial}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$(ms.name = \texttt{init})$$
$$m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods)$$
$$m' \neq \texttt{None}$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$Loc = \mathsf{getOneStackElem}(o, argCount)$$
$$Loc \neq \texttt{Null}$$
$$\mathsf{accessAllowedM}(m, m')$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getStackElemts}(o, argCount)$$
$$C1 = (ct = \texttt{Thread} \vee \texttt{Thread} \in \mathsf{allSuperClasses}(\mathcal{JE}, ct))$$
$$C2 = \mathsf{isLocation}(\mathsf{head}(l'')) \wedge \texttt{Runnable} \in \mathsf{allInterfaces}(\mathcal{JE}, \mathsf{getDynamicClass}(\mathcal{S}, \mathsf{head}(l'')))$$
$$C = C1 \wedge C2$$
$$\mathcal{S}' = \mathcal{S}[Loc \mapsto \mathcal{S}(Loc)[fromRunnable \leftarrow \mathsf{getDynamicClass}(\mathcal{S}, \mathsf{head}(l''))]]$$
$$o' = \mathsf{popStack}(o, argCount + 1)$$
$$nf = \mathsf{newFrame}(m', 0, Loc :: l'' :: l', [\,], \texttt{None})$$

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle$$

The second rule corresponds to the case where the method is either a private method or called using super keyword with no exceptions during the execution of the bytecode. This rule treats the case of non-synchronized methods.

$$m.code(pc) = \texttt{invokespecial}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$cm = m.fromClass$$
$$(ms.name \neq \texttt{init}) \wedge ((ct = cm) \vee (ct = \mathcal{JE}(cm).superClass))$$
$$C = (m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods)) \wedge (m' \neq \texttt{None}) \wedge (\mathsf{isPrivateM}(m'))$$
$$(ms.name \neq \texttt{init}) \wedge (ct = cm) \Rightarrow C$$
$$(ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass) \Rightarrow \mathsf{lookupM}(\mathcal{JE}(ct), ms, ct) \wedge m' \neq \texttt{None}$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$Loc = \mathsf{getOneStackElem}(o, argCount)$$
$$Loc \neq \texttt{Null}$$
$$\neg\, \mathsf{isSynchronized}(m') \wedge\ \mathsf{accessAllowedM}(m, m') \wedge\ \neg \mathsf{isStaticM}(m')$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getStackElemts}(o, argCount)$$
$$o' = \mathsf{popStack}(o, argCount + 1)$$
$$nf = \mathsf{newFrame}(m', 0, Loc :: l'' :: l', [\,], \texttt{None})$$

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle$$

The third case is the same as the latter except the fact that the method is synchronized. In this case, we consider that the monitor associated with the receiver can be acquired or reentered and is not owned by another thread. Contrarily to the previous rule, the store updates the monitor information of the gained object and if this latter was not already owned by the current thread, it will be added to the list of its locked elements. Furthermore, the field *synchronizedElement* of the new frame is set to the top of the operand stack of the current method.

$$m.code(pc) = \texttt{invokespecial}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$cm = m.fromClass$$
$$((ms.name \neq \texttt{init}) \wedge (ct = cm)) \vee ((ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass))$$
$$C = (m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods)) \wedge m'(\neq \texttt{None}) \wedge (\mathsf{isPrivateM}(m'))$$
$$(ms.name \neq \texttt{init}) \wedge (ct = cm) \Rightarrow C$$
$$(ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass) \Rightarrow \mathsf{lookupM}(\mathcal{JE}(ct), ms, ct) \wedge m' \neq \texttt{None}$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$Loc = \mathsf{getOneStackElem}(o, argCount)$$
$$\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \neg\mathsf{isStaticM}(m') \wedge Loc \neq \texttt{Null}$$
$$\neg\,\mathsf{isLocked}(\mathcal{S}, Loc) \vee \mathsf{isOwner}(\mathcal{S}, Loc, \iota)$$
$$\mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{objectMonitorEntered}(\mathcal{S}, Loc, \iota)]$$
$$l' = m'.methodVariables$$
$$l'' = \mathsf{getStackElemts}(o, argCount)$$
$$o' = \mathsf{popStack}(o, argCount + 1)$$
$$nf = \mathsf{newFrame}(m', 0, Loc :: l'' :: l', [\,], Loc)$$
$$\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isOwner}(\mathcal{S}, Loc, \iota), \mathcal{L}, Loc :: \mathcal{L})$$

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', nf :: \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None}\rangle$$

The fourth case is like the third one but where the monitor associated to the receiver is already owned by another thread. The current thread is added to the waiting list of the receiver (i.e., the object referenced by the top of the current thread's operand stack) thus updating the store. The transition label *block* transmits to the second layer the information that the current thread must be blocked.

$$m.code(pc) = \texttt{invokespecial}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$cm = m.fromClass$$
$$((ms.name \neq \texttt{init}) \wedge (ct = cm)) \vee ((ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass))$$
$$C = (m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods)) \wedge (m' \neq \texttt{None}) \wedge (\mathsf{isPrivateM}(m'))$$
$$(ms.name \neq \texttt{init}) \wedge (ct = cm) \Rightarrow C$$
$$(ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass) \Rightarrow \mathsf{lookupM}(\mathcal{JE}(ct), ms, ct) \wedge m' \neq \texttt{None}$$
$$argCount = \mathsf{length}(ms.argumentsTypes)$$
$$Loc = \mathsf{getOneStackElem}(o, argCount)$$
$$\mathsf{isSynchronized}(m') \wedge \mathsf{accessAllowedM}(m, m') \wedge \neg\mathsf{isStaticM}(m') \wedge Loc \neq \texttt{Null}$$
$$\mathsf{isLocked}(\mathcal{S}, Loc) \wedge \neg\, \mathsf{isOwner}(\mathcal{S}, Loc, \iota)$$
$$\mathcal{S}' = \mathcal{S}[Loc \mapsto \mathsf{addToObjectWaitingList}(\mathcal{S}, Loc, \iota)]$$
$$\overline{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \xrightarrow{block} \langle \mathcal{JE}, \mathcal{S}', \{|m, adr, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle}$$

The last following case is when exceptions occur while executing the `invokespecial` bytecode. In this case, a `Throwable` object is created in the store and its reference is assigned to the thread configuration exception flag.

$$m.code(pc) = \texttt{invokespecial}\ i$$
$$ms = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).methodSignature$$
$$ct = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass$$
$$cm = m.fromClass$$
$$C1 = (ms.name \neq \texttt{init}) \wedge (ct = cm) \wedge (m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods))$$
$$C2 = C1 \wedge ((m' = \texttt{None}) \vee (\neg\texttt{isPrivateM}(m')))$$
$$C3 = (ms.name = \texttt{init}) \wedge (m' = \mathsf{retrieveM}(ms, \mathcal{JE}(ct).methods)) \wedge (m' = \texttt{None})$$
$$C4 = (ms.name \neq \texttt{init}) \wedge (ct = \mathcal{JE}(cm).superClass) \wedge m' = \mathsf{Lookup}(\mathcal{JE}, ms, ct) \wedge m' = \texttt{None}$$
$$C = C2 \vee C3 \vee C4$$
$$Loc = \mathsf{getOneStackElem}(o, argCount)$$
$$C \vee \neg\, \mathsf{accessAllowedM}(m, m') \vee Loc = \texttt{Null} \vee \mathsf{isStaticM}(m') \vee \neg\mathsf{isMethResolved}(\mathcal{JE}, ms, ct)$$
$$\mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin Dom(\mathcal{S})$$
$$\overline{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None}\rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e\rangle}$$

**Method Return:** `return`

We present separately the semantics of void return which is reflected by the bytecode `return` and the return with values which is represented by the `ireturn` and `areturn` bytecodes. We will present only the rules of the `ireturn` since the rules describing the semantics of `areturn` are similar. We have split the `return` semantic rules in seven rules. The first one is when the method from which we return is a non-synchronized method. The six other cases are when the method is synchronized. In the first rule, the current method frame is popped from the frame stack and the program counter of its calling method is incremented.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \neg\ \mathsf{isSynchronized}(m) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The following rule refers to the case where the method is synchronized but not static. It considers that the current thread is still the owner of the object blocked when invoking the method.We remind the reader that the blocked object reference appears in the field $\mathcal{S}$ynchronizedElement of the frame and denoted by the variable $z$ in the following rule. We decrement the number of times this object has been locked but there is no need to notify the other threads waiting for the object because we consider here that the monitor's depth is not null after returning from the method. As for the first rule, the current method frame is popped from the frame stack and the program counter of its calling method is incremented.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \neg\ \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{JE}' = \mathcal{JE}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) \neq 0 \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The next rule describes the same case as the latter except that we notify the other threads waiting for the object referenced by the variable $z$. In fact, we consider in this rule that the object monitor's depth is null after returning from the method.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \neg\, \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{notify(z)} \langle \mathcal{JE}, \mathcal{S}', \{|n.pc' + 1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}$$

The following fourth rule refers to the case where the method is synchronized, not static but, contrarily to the two last rules, the current thread is not the owner of the object blocked when invoking the method. An exception is then thrown.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \neg\, \mathsf{isStaticM}(m) \wedge \neg\, \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})] \; ; \; e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

The three following rules represent the case where the return is done from a synchronized and static method. They are very close to the three last rules where the method was synchronized but not static.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{JE}' = \mathcal{JE}[z \mapsto \mathsf{classMonitorExited}(\mathcal{JE}, z, \iota)] \\ \mathsf{depthClassLock}(\mathcal{JE}', z) \neq 0 \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|n, pc' + 1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{classMonitorExited}(\mathcal{JE}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{notify(z)} \langle \mathcal{JE}, \mathcal{S}', \{|m, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{return} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \neg\, \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})] \; ; \; e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

**Method Return:** `ireturn`

The `ireturn` semantic rules are very similar to the `return` rules. Seven rules differentiate between the different cases where the method is not synchronized, synchronized and not static or synchronized and static. The only difference compared to the `return` rules is that there is a value in the operand stack of the current frame which is popped and pushed onto the operand stack of the frame of the invoker.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \neg\, \mathsf{isSynchronized}(m) \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \neg\, \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) \neq 0 \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \neg\, \mathsf{isStaticM}(m) \wedge \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \overset{notify(z)}{\longrightarrow} \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \neg\, \mathsf{isStaticM}(m) \wedge \neg\, \mathsf{isOwner}(\mathcal{S}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})] \;;\; e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{JE}' = \mathcal{JE}[z \mapsto \mathsf{classMonitorExited}(\mathcal{JE}, z, \iota)] \\ \mathsf{depthClassLock}(\mathcal{JE}', z) \neq 0 \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{JE}' = \mathcal{JE}[z \mapsto \mathsf{objectMonitorExited}(\mathcal{S}, z, \iota)] \\ \mathsf{depthLock}(\mathcal{S}', z) = 0 \\ \mathcal{L}' = \mathsf{suppress}(z, \mathcal{L}) \\ o' = \mathsf{pushStack}(o', \mathsf{head}(o)) \\ f = \{|n, pc', l', o', z'|\} \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: f :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{notify(z)} \langle \mathcal{JE}, \mathcal{S}', \{|n, pc'+1, l', o', z'|\} :: \mathcal{F}, \mathcal{L}', \iota, \texttt{None} \rangle}$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{ireturn} \\ \mathsf{isSynchronized}(m) \wedge \mathsf{isStaticM}(m) \wedge \neg\, \mathsf{isClassOwner}(\mathcal{JE}, z, \iota) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})] \; ; \; e \notin Dom(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

**Field Access Statements**

This section describes the semantic rules of `getfield`, `putfield`, `getstatic` and `putstatic` instructions. The first rule refers to the semantics of the `getfield` bytecode when no exceptions are signaled. The exceptions can be thrown either when a reference on the top of the operand stack is null, or when the considered field is not found or not accessible or static. In absence of exceptions, the following rule is applied and the object reference on the top of the operand stack is popped and the value of the considered field in the object referenced is fetched and pushed onto the operand stack.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{getfield}\, i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ Loc = \mathsf{getOneStackElem}(o, 0) \\ (Loc \neq \texttt{Null}) \wedge (f \neq \texttt{None}) \wedge (\mathsf{accessAllowedF}(f, m)) \wedge (\neg\, \mathsf{isStaticF}(f)) \\ o' = \mathcal{S}(Loc).\mathit{fieldMap}(f) :: \mathsf{popStack}(o, 1) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The following rule refers to the semantics of the `getfield` bytecode in presence of one of the exceptions cited previously. A new exception is then thrown.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{getfield}\, i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ \mathit{Loc} = \mathsf{getOneStackElem}(o, 0) \\ (f = \texttt{None}) \vee (\mathit{Loc} = \texttt{Null}) \vee (\neg\ \mathsf{accessAllowedF}(f, m)) \vee \mathsf{isStaticF}(f) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin \mathit{Dom}(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

The next rule refers to the semantics of the `putfield` bytecode when no exceptions are signaled. The exceptions that can be thrown are similar to those of `getfield`. In this case, two elements are popped from the operand stack: A value and an object reference. The considered field in the object reference is then set to the popped value.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{putfield}\, i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ v = \mathsf{getOneStackElem(o,0)} \\ \mathit{Loc} = \mathsf{getOneStackElem(o,1)} \\ (f \neq \texttt{None}) \wedge (\mathit{Loc} \neq \texttt{Null}) \wedge (\mathsf{accessAllowedF}(f, m)) \wedge (\neg\ \mathsf{isStaticF}(f)) \\ \mathcal{S}' = \mathcal{S}[\mathit{Loc} \mapsto \mathcal{S}(\mathit{Loc})[\mathit{fieldsMap} \leftarrow \mathcal{S}(\mathit{Loc}).\mathit{fieldsMap}[f \mapsto v]]] \\ o' = \mathsf{posStack}(o, 2) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

The following rule refers to the semantics of the `putfield` bytecode in presence of exceptions and an exception is then thrown.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{putfield}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ \mathit{Loc} = \mathsf{getOneStackElem(o,1)} \\ (f = \texttt{None}) \vee (\mathit{Loc} = \texttt{Null}) \vee (\neg\ \mathsf{accessAllowedF}(f, m)) \vee \mathsf{isStaticF}(f) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin \mathit{Dom}(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, e \rangle}$$

The four following instructions refer to `getstatic` semantics. The first rule describes the semantics in the case where the field to read is found, accessible and static. This rule considers also that the class of the considered field has been initialized. The value of the class or the interface field is then fetched and pushed onto the operand stack.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{getstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ cf = f.\mathit{fromClass} \\ (f \neq \texttt{None}) \wedge (\mathsf{accessAllowedF}(f, m)) \wedge (\mathsf{isStaticF}(f)) \wedge (\mathsf{isInitialized}(\mathcal{JE}, cf) \\ o' = \mathcal{JE}(cf).\mathit{staticMap}(f) :: o \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{t}, \texttt{None} \rangle}$$

The next rule describes the semantics in case the field to read is found but its corresponding class has not been initialized. In this rule, we consider that the class is not locked by another thread and a frame of its `clinit` method is pushed onto the frame stack. The class is locked and then added to the locked elements of the thread if it is not there yet.

$$
\frac{\begin{array}{c} m.code(pc) = \texttt{getstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).fieldSignature \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ f \neq \texttt{None} \\ cf = f.fromClass \\ \neg\ \mathsf{isInitialized}(\mathcal{JE}, cf) \\ \neg\ \mathsf{isClassLocked}(\mathcal{JE}, cf) \vee \mathsf{isClassOwner}(\mathcal{JE}, cf, \iota) \\ signatureClinit = \langle \texttt{clinit}, [\,], \texttt{void} \rangle \\ clinit = \mathsf{retrieveM}(signatureClinit, \mathcal{JE}(cf).methods) \\ clinitFrame = \mathsf{newFrame}(clinit, 0, clinit.methodVariables, [\,], cf) \\ \mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{JE}, cf, \iota), \mathcal{L}, cf :: \mathcal{L}) \\ \mathcal{JE}' = \mathcal{JE}[cf \mapsto \mathsf{classMonitorEntered}(\mathcal{JE}, cf, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, clinitFrame :: \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}
$$

The third rule refers also to the case where the field to read is found and its corresponding class has not been initialized yet. The difference from the last rule is that another thread owns the monitor of this class. The current thread is then blocked waiting for this monitor.

$$
\frac{\begin{array}{c} m.code(pc) = \texttt{getstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).fieldSignature \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).supposedClass \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ f \neq \texttt{None} \\ cf = f.fromClass \\ \neg\ \mathsf{isInitialized}(\mathcal{JE}, cf) \\ \mathsf{isClassLocked}(\mathcal{JE}, cf) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{JE}, cf, \iota) \\ \mathcal{JE}' = \mathcal{JE}[cf \mapsto \mathsf{addToClassWaitingList}(\mathcal{JE}, cf, \iota)] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{block} \langle \mathcal{JE}', \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}
$$

The last rule for the `getstatic` bytecode refers to the case of exceptions. An exception is thrown if either the field is not found, or not accessible or not static.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{getstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ (f = \texttt{None}) \vee (\neg\ \mathsf{accessAllowedF}(f, m)) \vee (\neg\ \mathsf{isStaticF}(f)) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin \mathit{Dom}(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, e \rangle}$$

The instruction `putstatic` allows in normal cases to pop a value from the operand stack and to set the considered class field to this value. Four rules describe the semantics of the `putstatic` instruction and are very close to the `getstatic` ones.

$$\frac{\begin{array}{c} m.code(pc) = \texttt{putstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ cf = f.\mathit{fromClass} \\ (f \neq \texttt{None}) \wedge (\mathsf{accessAllowedF}(f, m)) \wedge (\mathsf{isStaticF}(f)) \wedge (\mathsf{isInitialized}(\mathcal{JE}, cf) \\ v = \mathsf{getOneStackElem}(o, 0) \\ o' = \mathsf{popStack}(o, 1) \\ \mathcal{JE}' = \mathcal{JE}[ct \mapsto \mathcal{JE}(ct)[\mathit{staticMap} \leftarrow \mathcal{JE}(ct).\mathit{staticMap}[f \mapsto v]]] \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, \{|m, pc+1, l, o', z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle}$$

$$m.code(pc) = \texttt{putstatic}\ i$$
$$fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature}$$
$$c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass}$$
$$f = \mathsf{lookupF}(\mathcal{JE}, fs, c)$$
$$f \neq \texttt{None}$$
$$cf = f.\mathit{fromClass}$$
$$\neg\ \mathsf{isInitialized}(\mathcal{JE}, cf)$$
$$\neg\ \mathsf{isClassLocked}(\mathcal{JE}, cf) \vee \mathsf{isClassOwner}(\mathcal{JE}, cf, \iota)$$
$$\mathit{signatureClinit} = \langle \texttt{clinit}, [\,], \texttt{void} \rangle$$
$$\mathit{clinit} = \mathsf{retrieveM}(\mathit{signatureClinit}, \mathcal{JE}(cf).\mathit{methods})$$
$$\mathit{clinitFrame} = \mathsf{newFrame}(\mathit{clinit}, 0, \mathit{clinit}.\mathit{methodVariables}, [\,], cf)$$
$$\mathcal{L}' = \mathsf{ifThenElse}(\mathsf{isClassOwner}(\mathcal{JE}, cf, \iota), \mathcal{L}, cf :: \mathcal{L})$$
$$\mathcal{JE}' = \mathcal{JE}[cf \mapsto \mathsf{classMonitorEntered}(\mathcal{JE}, cf, \iota)]$$

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}', \mathcal{S}, \mathit{clinitFrame} :: \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle$$

$$m.code(pc) = \texttt{putstatic}\ i$$
$$fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature}$$
$$c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass}$$
$$f = \mathsf{lookupF}(\mathcal{JE}, fs, c)$$
$$f \neq \texttt{None}$$
$$cf = f.\mathit{fromClass}$$
$$\neg\ \mathsf{isInitialized}(\mathcal{JE}, cf)$$
$$\mathsf{isClassLocked}(\mathcal{JE}, cf) \wedge \neg\ \mathsf{isClassOwner}(\mathcal{JE}, cf, \iota)$$
$$\mathcal{JE}' = \mathcal{JE}[cf \mapsto \mathsf{addToClassWaitingList}(\mathcal{JE}, cf, \iota)]$$

$$\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle \xrightarrow{\mathit{block}} \langle \mathcal{JE}', \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \iota, \texttt{None} \rangle$$

$$\frac{\begin{array}{c} m.code(pc) = \texttt{putstatic}\ i \\ fs = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{fieldSignature} \\ c = \mathsf{thisConstantPoolEntry}(\mathcal{JE}, m, i).\mathit{supposedClass} \\ f = \mathsf{lookupF}(\mathcal{JE}, fs, c) \\ (f = \texttt{None}) \vee (\neg\ \mathsf{accessAllowedF}(f, m)) \vee (\neg\ \mathsf{isStaticF}(f)) \\ \mathcal{S}' = \mathcal{S}[e \mapsto \mathsf{newObject}(\mathcal{JE}, \texttt{Throwable})]\ ;\ e \notin \mathit{Dom}(\mathcal{S}) \end{array}}{\langle \mathcal{JE}, \mathcal{S}, \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{l}, \texttt{None} \rangle \longrightarrow \langle \mathcal{JE}, \mathcal{S}', \{|m, pc, l, o, z|\} :: \mathcal{F}, \mathcal{L}, \mathfrak{l}, e \rangle}$$

### 4.3.2 Second Layer

We present, in this section, all the rules of the second layer. There are five different rules, one for each transition label. The first rule illustrates the case when an ε-transition is done in the first layer. In this case, we have only to report in the Java stack the fact that the current thread has changed its information from $\mathcal{T}$ to $\mathcal{T}'$ maintaining its state to `active`. The new Java stack $\mathcal{JS}'$ is identical to the initial Java stack $\mathcal{JS}$ except for the current thread.

$$\frac{\begin{array}{c} \langle \mathcal{JE},\ \mathcal{S},\ \mathcal{T} \rangle \longrightarrow \langle \mathcal{JE}',\ \mathcal{S}',\ \mathcal{T}' \rangle \\ \mathcal{T}'.\mathit{exception} = \texttt{None} \\ \mathcal{JS}' = \mathsf{changeThreads}(\mathcal{JS}, \mathcal{T}, \mathcal{T}') \end{array}}{\langle \mathcal{JE},\ \mathcal{S},\ \mathcal{JS} \rangle\!\!>\!\!\hookrightarrow \langle \mathcal{JE}',\ \mathcal{S}',\ \mathcal{JS}' \rangle}$$

The next rule is executed when the transition in the first layer is done via the label *block*. In this case, the Java stack has to change also the state of the current thread from `active` to `blocked` because it asks for a resource that is not available.

$$\frac{\begin{array}{c} \langle \mathcal{JE},\ \mathcal{S},\ \mathcal{T} \rangle \xrightarrow{\mathit{block}} \langle \mathcal{JE}',\ \mathcal{S}',\ \mathcal{T}' \rangle \\ \mathcal{JS}' = \mathsf{blockThreads}(\mathcal{JS}, \mathcal{T}) \end{array}}{\langle \mathcal{JE},\ \mathcal{S},\ \mathcal{JS} \rangle \hookrightarrow \langle \mathcal{JE}',\ \mathcal{S}',\ \mathcal{JS}' \rangle}$$

The following rule defines the configuration transition in the second layer when the transition in the first layer is done via the label *notify*. In this case, the current thread has released an object or a class $x$ and all the threads that are waiting for this resource $x$ must be now `active`.

$$\frac{\langle \mathcal{JE}, S, \mathcal{T}\rangle \xrightarrow{notify(x)} \langle \mathcal{JE}', S', \mathcal{T}'\rangle \quad \mathcal{JS}' = \mathsf{changeThreads}(\mathcal{JS}, \mathcal{T}, \mathcal{T}') \quad \mathcal{JS}'' = \mathsf{activateThreads}(\mathcal{JS}', S, \mathcal{JE}, [x])}{\langle \mathcal{JE}, S, \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{JE}', S', \mathcal{JS}''\rangle}$$

The next rule presents the configuration transition in the second layer in presence of a transition in the first layer done with the label *kill*. In this case, the current thread throws an exception that is not caught by any method along its method invocation stack. The thread will then expire but it will first activate all the threads waiting for its locked objects.

$$\frac{\langle \mathcal{JE}, S, \mathcal{T}\rangle \xrightarrow{kill} \langle \mathcal{JE}', S', \mathcal{T}'\rangle \quad \mathcal{JS}' = \mathsf{activateThreads}(\mathcal{JS}, S, \mathcal{JE}, \mathcal{T}.lockedElements) \quad \mathcal{JS}'' = \mathsf{dieThread}(\mathcal{JS}', \mathcal{T}.threadId)}{\langle \mathcal{JE}, S, \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{JE}', S', \mathcal{JS}''\rangle}$$

Finally, the last rule describes the configuration transition in the second layer when the transition in the first layer is done via the label *run*. In this case, a new thread is created and starts its execution by the adequate run method given by the lookup.

$$\frac{\langle \mathcal{JE}, S, \mathcal{T}\rangle \xrightarrow{run(class)} \langle \mathcal{JE}', S', \mathcal{T}'\rangle \quad signatureRun = \langle run, [\,], \mathtt{void}\rangle \quad run = \mathsf{lookupM}(\mathcal{JE}, signatureRun, class) \quad f = \mathsf{newFrame}(run, 0, run.methodVariables, [\,], \mathtt{None}) \quad \mathcal{JS}' = \mathcal{JS}[\iota \mapsto \mathsf{newThreadInformation}([f], [\,], \iota)\ ;\ \iota \notin Dom(\mathcal{JS})}{\langle \mathcal{JE}, S, \mathcal{JS}\rangle \hookrightarrow \langle \mathcal{JE}', S', \mathcal{JS}'\rangle}$$

## 4.4 Conclusion

In this chapter, we reported a formalization of the dynamic semantics of JVML. The semantics comes into a small step operational style. In order to ascribe meanings to threading, the semantics is structured in two layers: The first layer capture the semantics of sequential JVML programs in isolation. The second layer consists of judgements that capture the parallel execution of JVML threads. A nice feature of the presented semantics is its faithfulness to the official JVML specification. Besides, the presented

semantics provides full account details for the most technical and tricky aspects of JVML such as multi-threading, synchronization, method invocations, exception handling, object creation, objectŠs fields manipulation, stack manipulation, local variable access, modifiers, etc. The presented semantics is also, to the best of our knowledge, the first dynamic semantics of JVML that provides semantics for that many features within the same framework.

In the next chapter, we will present the semantics of AspectJ advice weaving, which is performed on JVML codes.

# Chapter 5

# AspectJ Weaving Semantics

This chapter presents a formal semantics that describes the advice weaving in AspectJ. It compiles the know-how of the AspectJ community into an elegant semantic framework. In order to build this semantics, we had to delve into the Eclispe AspectJ compiler code source (`ajc` 1.2). This task was tedious and time consuming because it required reading thousands of lines of code without any design documentation. We had to scrutinize both the source code of programs and the corresponding compiled units in order to determine how the AspectJ primitives are interpreted by the compiler. To build the semantics, we first formalize the pointcuts and the shadows description. Afterwards, we formally define the AspectJ environment that contains the program declarations. This latter is an extension of the JVML environment defined in Tables 4.4 and 4.5. Thirdly, we describe the matching process that allows to decide wether a given join point matches a given pointcut. The pointcut matching process differs depending on if the pointcut is static or dynamic. Static pointcuts can be directly mapped to code and the matching process knows at compile time if a join point matches a given pointcut or not. Contrary to static pointcuts, dynamic pointcuts cannot be definitely mapped to places in code and the matching process often calculates, for each join point and dynamic pointcut, the tests (called residues)

to be executed at runtime in order to check if the join point matches the dynamic properties of the pointcut. Hence, the output of the pointcut matching process is a set of matched join points where each matched join point is accompanied with the set of advices that are going to be inserted at this join point with the corresponding tests, if any. We describe also the JVML instructions that are generated for these dynamic tests. Finally, we give the semantic rules of the advice weaving.

## 5.1 Pointcuts, Join Points, and Shadows

In our contribution, we consider the following base pointcuts: "method call", "constructor call", "method execution", "constructor execution", "advice execution", "within method code", "field get", "field set", "static class initialization", "within class", "this", "target", and "args". These pointcuts can be logically combined, using boolean operations, to construct more complex ones.

Table 5.1 formally defines these pointcuts. The type *ComponentType* in this table represents the Java classes and the aspects. Java classes are described by the type *ReferenceType* and the aspects by the type *AspectType*. In Chapter 4, the type *Type* contains primitive types and reference types. In this chapter, we add also the aspects to the type *Type*. The type *MethodPattern* used in some of the pointcuts is a combination of a set of method modifiers, a method signature, and a component type where the considered method occurs. In the same way, the type *ContructPattern* describes constructors.

The set of join points and shadows where advice weaving may intervene is given in Table 5.2. As we can notice, a shadow of a method call may be `invokevirtual`, `invokestatic`, `invokespecial`, or `invokeinterface` instruction. This will depend on the nature of the method. For example, if the method is static the shadow must be an `invokestatic` instruction. Furthermore, there are two cases of join point shadows: The case where the shadow is exactly one instruction and the case where the shadow is an entire method. We can notice also that the pointcuts "within method code",

| | | | |
|---|---|---|---|
| *Pcut* | ::= | *BasePcut* | \| *BooleanPcut* |
| *BasePcut* | ::= | `call(`*Pattern*`)` | \| `execution(`*Pattern*`)` |
| | \| | `aexecution()` | \| `args((`*Type*`)-list)` |
| | \| | `get(`*FieldPattern*`)` | \| `set(`*FieldPattern*`)` |
| | \| | `staticinit(`*ComponentType*`)` | \| `within(`*ComponentType*`)` |
| | \| | `this(`*ComponentType*`)` | \| `target(`*ComponentType*`)` |
| | \| | `withincode(`*MethodPattern*`)` | |
| *BooleanPcut* | ::= | *Pcut* `or` *Pcut* \| `not` *Pcut* \| *Pcut* `and` *Pcut* | |
| *Pattern* | ::= | *MethodPattern* \| *ConstructPattern* | |
| *MethodPattern* | ::= | ⟨*methodModifiers*: (*MethodModifier*)`-set`, *methodSignature*: *MethodSignature*, *componentType*: *ComponentType* ⟩ | |
| *ConstructPattern* | ::= | ⟨*constructModifiers*: (*ConstructModifier*)`-set`, *argumentsType*: (*Type*)`-list`, *componentType*: *ClassType* ⟩ | |
| *ConstructModifier* | ::= | `public \| private` | |
| *FieldPattern* | ::= | ⟨*fieldModifiers*: (*FieldModifier*)`-set`, *fieldSignature*:*FieldSignature*, *componentType*: *ComponentType* ⟩ | |
| *Type* | ::= | *PrimitiveType*\|*ComponentType* | |
| *ComponentType* | ::= | *ReferenceType* \| *AspectType* | |
| *AspectType* | ::= | *Identifier* | |

Table 5.1: Pointcuts

"within class", "this", "target", and "args" do not by themselves define new shadows. They use the shadows defined by the other six static pointcuts.

Table 5.3 describes formally shadows with the type *Shadow*. It is a combination of the shadow kind, its signature, its modifiers, the class that is going to be involved by the shadow, two natural numbers that represent the start and the end of the shadow and a list of shadow mungers. The kind of a shadow is represented by a pair of a name and a boolean. The boolean indicates if the arguments of this shadow are on the execution stack or not. Thi is necessary to know if we have to pop the arguments on the stack or not when injecting the advices and conditional tests. The shadow mungers are an abstract representation of the advices and the tests to be injected into their associated shadow. The different possible values of a shadow munger test are the following:

| JOIN POINT | SHADOW |
|---|---|
| Method call | `invokevirtual` *i*<br>`invokespecial` *i* (for private methods)<br>`invokestatic` *i*<br>`invokeinterface` *i*,*n* |
| Constructor call | `invokespecial` *i* |
| Field get | `getfield` *i*<br>`getstatic` *i* |
| Field set | `putfield` *i*<br>`putstatic` *i* |
| Method execution | Entire method code |
| Constructor execution | Entire "init" code |
| Advice execution | Entire advice code |
| Static initialization | Entire "clinit" code |

Table 5.2: Join Points and Shadows.

- `always`: The advice is injected without JVML test instructions because the advice pointcut always matches the shadow.
- `never`: The advice is not injected since its pointcut never matches the shadow.
- `target instanceof(`*c*`)`: JVML instructions that test if the target object is an instance of *c* are injected with the advice.
- `arg(`*i*`) instanceof(`*c*`)`: JVML instructions that test if the argument *i* is an instance of *c* are injected with the advice.
- `Not(`*t*`)`: JVML instructions testing if *t* is not satisfied are injected with the advice.
- `And(`$t_1$`,`$t_2$`)`: JVML instructions testing if $t_1$ and $t_2$ are satisfied are injected with the advice.
- `Or(`$t_1$`,`$t_2$`)`: JVML instructions testing if $t_1$ or $t_2$ are satisfied are injected with the advice.

```
Shadow    ::= ⟨kind: Kind,
              signature: Signature,
              modifiers: ( MethodModifier) -set|( FieldModifier) -set,
              fromClass: ComponentType,
              start: Nat,
              end: Nat,
              mungers: (ShadowMunger) -list ⟩
Kind          ::= (method_call,true)
                | (method_execut,false)
                | (field_get,true)
                | (field_set,true)
                | (advice_execut,false)
                | (construct_call,true)
                | (construct_execut,false)
                | (static_init,false)
Signature     ::= MethodSignature | FieldSignature
ShadowMunger  ::= ⟨adviceInfo: AdviceInfo,
                  pointcutTest: Test ⟩
Test          ::= always | never
                | this instanceof(ComponentType )
                | target instanceof(ComponentType )
                | arg(Nat) instanceof(ComponentType )
                | Not(Test) | And(Test,Test) | Or(Test,Test)
```

Table 5.3: Shadow Syntax

In the same context, we define the following functions makeAnd, makeOr, and makeNot that build new tests resulting respectively from conjunction, disjunction, and negation of existing tests. These functions are necessary to generate combined tests when matching a combined pointcut with a given join point shadow. The formal definitions of these functions are the following:

makeAnd: *Test*×*Test* → *Test*

makeAnd(`never`,*b*)=`never`

makeAnd(*a*,`never`)=`never`

makeAnd(*a*,`always`)=*a*

makeAnd(`always`,*b*)=*b*

makeAnd(*a*,*b*)=`And`(a,b) otherwise

makeOr: *Test*×*Test* → *Test*

makeOr(`always`,*b*)=`always`

makeOr(*a*,`always`)=`always`

makeOr(*a*,`never`)=*a*

makeOr(`never`,*b*)=*b*

makeOr(*a*,*b*)=`Or`(a,b) otherwise

makeNot: *Test* → *Test*

makeNot(`always`)=`never`

makeNot(`never`)=`always`

makeNot(*a*)=`Not`(a) otherwise

Two functions named shadowing and preShadowing are presented in Appendix I and allow us to calculate the different shadows in a method. The function preShadowing calculates all the execution shadows ("method execution", "constructor execution", "advice execution", and "static class initialization") and the function shadowing calculates

all the non-execution shadows. These functions change the method codes by inserting `impdep1` to mark the start and the end of shadows. In the example of Figure 5.1, the functions shadowing and preShadowing will insert in the JVML code of $f$ four pairs of `impdep1` codes to mark the four shadows of the method, which are: The execution of $f$, the call to $g$, the get field of $b$ and finally the set field of $a$. The records corresponding to the four shadows are as following:

| **Code Java** | **Initial JVML code of $f$** | **JVML code of $f$ after shadowing** |
|---|---|---|
| `class C {`<br>`int a,b;`<br>`public void f(){`<br>`int c=4;`<br>`g();`<br>`a=b;`<br>`}`<br>`private void g(){`<br>`}`<br>`}` | 1: `aload_0`<br>2: `invokespecial #2`<br>3: `aload_0`<br>4: `aload_0`<br>5: `getfield #3`<br>6: `putfield #4`<br>7: `return` | 1: `impdep1`<br>2: `aload_0`<br>3: `impdep1`<br>4: `invokevirtual #2`<br>5: `impdep1`<br>6: `aload_0`<br>7: `aload_0`<br>8: `impdep1`<br>9: `getfield #3`<br>10: `impdep1`<br>11: `impdep1`<br>12: `putfield #4`<br>13: `impdep1`<br>14: `return`<br>15: `impdep1` |

Figure 5.1: Shadowing Example

1. $\langle \texttt{method\_execut}, \langle f, [\,], \texttt{void} \rangle, \{\texttt{public}\}, C, 1, 15, [\,] \rangle$
2. $\langle \texttt{method\_call}, \langle g, [\,], \texttt{void} \rangle, \{\texttt{private}\}, C, 3, 5, [\,] \rangle$
3. $\langle \texttt{field\_get}, \langle b, int \rangle, \{\}, C, 8, 10, [\,] \rangle$
4. $\langle \texttt{field\_set}, \langle a, int \rangle, \{\}, C, 11, 13, [\,] \rangle$

## 5.2 Environnement

We describe in Table 5.4 the AspectJ environment. It is represented as a record containing a Java environment and a set of advices. The Java environment is quite similar to the environment of Table 4.4 and Table 4.5 and is a map that associates a set of classes to a set of component types. However, the component type is now considered to be a reference type (interface or class) or an aspect. We consider that an aspect can be viewed as a class and its advices are represented by the methods of the class. A new flag is also added to the class record indicating wether the component is an aspect or not. We also consider that an instruction may be either a JVML instruction or a code `impdep1`. Before starting the pointcut matching process, AspectJ inserts the mnemonic `impdep1` before and after all the possible join point shadows. The pointcut matching process will then operate only on those "marked" regions. An advice is represented by a record containing its kind, its pointcut, the aspect where the advice has been defined, and its signature. Concerning the kind of advices, we examine both before advices and after advices. The after advice runs after the normal completion of a join point.

## 5.3 Matching Process

AspectJ weaving is essentially based on the matching process that calculates for each shadow the list of advices that it matches. In this section, we will describe the matching process in AspectJ with the function: matchpcut. This function takes an environment, a method, a shadow and a pointcut and returns a test condition under which the shadow satisfies the pointcut. We can notice that the resulting dynamic test from matching a shadow with a combined pointcut is a combination of the dynamic tests resulting when matching each basic pointcut with the shadow. We can also remark that in the case of a pointcut `target`(*ct*) for example, a test of the form: `target instanceof`(*ct*) is generated

| | | |
|---|---|---|
| *Environment* | ::= | ⟨ *javaEnvironment: JavaEnvironment*, |
| | | *advices:* (*AdviceInfo*) `-list` ⟩ |
| *JavaEnvironment* | ::= | *ComponentType* $\underset{m}{\rightarrow}$ *Class* |
| *Class* | ::= | ⟨ *constantPool:ConstantPool*, |
| | | *superClass: ClassType* \| *NoneType*, |
| | | *interfaces:* (*ClassType*)`-set`, |
| | | *fields:* (*Field*) `-list`, |
| | | *staticMap: Field* $\underset{m}{\rightarrow}$ *Value*, |
| | | *methods:* (*Method*) `-list`, |
| | | *initialized: Nat*, |
| | | *interface: Nat*, |
| | | *aspect: Nat*⟩ |
| *Code* | ::= | *ProgramCounter* $\underset{m}{\rightarrow}$ *Instruction* |
| *Instruction* | ::= | *JVMLInst* \| `impdep1` |
| *AdviceInfo* | ::= | ⟨*akind:* {`Before`, `After`}, |
| | | *pointcut: Pcut*, |
| | | *fromClass: AspectType*, |
| | | *adviceSignature: MethodSignature* ⟩ |

Table 5.4: AspectJ Environnement

when the shadow has a super class of *ct* as target. We can also easily check that a pointcut `args` never matches with a get field join point. The function matchPcut is defined formally hereafter.

matchPcut : $Environment \times Method \times Shadow \times Pointcut \rightarrow Test$

matchPcut($\mathcal{E}$,*m*,*s*,*pcut*1 `and` *pcut*2)=

  makeAnd(matchPcut($\mathcal{E}$,*m*,*s*,*pcut*1), matchPcut($\mathcal{E}$,*m*,*s*,*pcut*2))

matchPcut($\mathcal{E}$,*m*,*s*,*pcut*1 `or` *pcut*2)=

  makeOr(matchPcut($\mathcal{E}$,*m*,*s*,*pcut*1),matchPcut($\mathcal{E}$,*m*,*s*,*pcut*2)

matchPcut($\mathcal{E}$,*m*,*s*,`not` *pcut*) =makeNot( matchPcut($\mathcal{E}$,*m*,*s*,*pcut*))

matchPcut($\mathcal{E}$,$m$,$s$,`call`($p$))=`always`

$$\text{if}\begin{cases} & p.\mathit{methodSignature.name} \neq \texttt{init} \\ \wedge & \pi 1(s.\mathit{kind}) = \texttt{method\_call} \\ \wedge & s.\mathit{signature} = p.\mathit{methodSignature} \\ \wedge & s.\mathit{modifiers} \supseteq p.\mathit{methodModifiers} \\ \wedge & s.\mathit{fromClass} = p.\mathit{componentType} \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`call`($p$))=`always`

$$\text{if}\begin{cases} & p.\mathit{methodSignature.name} = \texttt{init} \\ \wedge & \pi 1(s.\mathit{kind}) = \texttt{construct\_call} \\ \wedge & s.\mathit{signature} = p.\mathit{methodSignature} \\ \wedge & s.\mathit{modifiers} \supseteq p.\mathit{methodModifiers} \\ \wedge & s.\mathit{fromClass} = p.\mathit{componentType} \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`call`($p$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`execution`($p$)) =`always`

$$\text{if}\begin{cases} & p.\mathit{methodSignature.name} \neq \texttt{init} \\ \wedge & \pi 1(s.\mathit{kind}) = \texttt{method\_execut} \\ \wedge & s.\mathit{signature} = p.\mathit{methodSignature} \\ \wedge & s.\mathit{modifiers} \supseteq p.\mathit{methodModifiers} \\ \wedge & s.\mathit{fromClass} = p.\mathit{componentType} \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`execution`($p$)) =`always`

$$\text{if}\begin{cases} & p.\mathit{methodSignature.name} = \texttt{init} \\ \wedge & \pi 1(s.\mathit{kind}) = \texttt{construct\_execut} \\ \wedge & s.\mathit{signature} = p.\mathit{methodSignature} \\ \wedge & s.\mathit{modifiers} \supseteq p.\mathit{methodModifiers} \\ \wedge & s.\mathit{fromClass} = p.\mathit{componentType} \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`execution`($p$)) =`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`staticinit`($ct$))=`always`

$$\text{if} \begin{cases} & \pi 1(s.kind) = \texttt{static\_init} \\ \wedge & s.fromClass = ct \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`staticinit`($ct$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`aexecution()`)=`always`

if $\pi 1(s.kind) = \texttt{advice\_execut}$

matchPcut($\mathcal{E}$,$m$,$s$,`aexecution()`)=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`get`($fp$))=`always`

$$\text{if} \begin{cases} & \pi 1(s.kind) = \texttt{field\_get} \\ \wedge & s.signature = fp.fieldSignature \\ \wedge & s.modifiers \supseteq fp.fieldModifiers \\ \wedge & s.fromClass = fp.componentType \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`get`($fp$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`set`($fp$))=`always`

$$\text{if} \begin{cases} & \pi 1(s.kind) = \texttt{field\_set} \\ \wedge & s.signature = fp.fieldSignature \\ \wedge & s.modifiers \supseteq fp.fieldModifiers \\ \wedge & s.fromClass = fp.componentType \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`set`($fp$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`within`($ct$))=`always` if $m.fromClass = ct$

matchPcut($\mathcal{E}$,$m$,$s$,`within`($ct$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`withincode`($mp$))=`always`

$$\text{if} \begin{cases} & m.methodModifiers \supseteq mp.methodModifiers \\ \wedge & m.methodSignature = mp.methodSignature \\ \wedge & m.fromClass = mp.ComponentType \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`withincode`($mp$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`this`($ct$))=`always`

$$\text{if} \begin{cases} & \mathsf{hasThis}(s,m) \\ \wedge & m.\mathit{fromClass} = ct \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`this`($ct$))=`this instanceof`($ct$)

$$\text{if} \begin{cases} & \mathsf{hasThis}(s,m) \\ \wedge & m.\mathit{fromClass} \in \mathsf{allSuperclasses}(\mathcal{E},\{ct\}) \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`this`($ct$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`target`($ct$))=`always`

$$\text{if} \begin{cases} & \mathsf{hasTarget}(s,m) \\ \wedge & \texttt{static} \notin s.\mathit{modifiers} \\ \wedge & s.\mathit{fromClass} = ct \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`target`($ct$))=`target instanceof`($ct$)

$$\text{if} \begin{cases} & \mathsf{hasTarget}(s,m) \\ \wedge & \texttt{static} \notin s.\mathit{modifiers} \\ \wedge & s.\mathit{fromClass} \in \mathsf{allSuperclasses}(\mathcal{E},\{ct\}) \end{cases}$$

matchPcut($\mathcal{E}$,$m$,$s$,`target`($ct$))=`never` otherwise

matchPcut($\mathcal{E}$,$m$,$s$,`args`($l$))=

argSTest($\mathcal{E}$,1,$s.\mathit{signature.argumentsType}$,$l$)

if $\pi 1(s.\mathit{kind}) \notin$ {`field_get`,`field_set`,`static_init` }

matchPcut($\mathcal{E}$,$m$,$s$,`args`($l$))=argTest($\mathcal{E}$,1,$s.\mathit{signature.type}$,head($l$))

if $\pi 1(s.\mathit{kind}) \in$ {`field_set` }

matchPcut($\mathcal{E}$,$m$,$s$,`args`($l$))=`never` otherwise

## 5.4 JVML Codes of Dynamic Tests

In this section, we describe the JVML codes corresponding to the dynamic tests previously presented. A dynamic test is composed of elements of the form:

"`this instanceof(`*c*`)`", "`target instanceof(`*c*`)`", and "`arg(`*i*`) instanceof(`*c*`)`".

In order to generate the JVML code for a dynamic test, we need to know the location, in the method local variables, of the objets to test (this, target, and argument). This is not a problem for the executing object (this) because we know that it is always at position 0 in the local variables. Hence an instruction `aload_0` is enough to load it onto the operand stack. However, we need to know the location of the target and arguments of the shadow. The AspectJ compiler stores the arguments and the target (if any) of a shadow in temporary local variables in the method local variables table. These temporary variables are loaded onto the operand stack and tested before injecting the advice. AspectJ compiler differentiates between the case where the current shadow has its arguments and its target (if any) on the stack and the case where it has not. In the case where they are on the stack, the compiler stores them in temporary local variables. In the other case, the compiler will load the arguments and the target (if any) from their initial locations in the method local variables table and store them in temporary local variables. We formalized this with two functions, named respectively insertBeforeStore and insertAfterStore presented in Appendix II. If the shadow has its arguments on the stack then the function returns only store instructions in order to store the arguments and the target in temporary variables. If the shadow does not have its arguments on the stack then the function returns load and store instructions because the arguments should be loaded from their original emplacement in the local variables table and restore them in temporary variables.

The function getTestInstructions generates the JVML instructions that correspond to a given dynamic test. Given an initial environment, a method, a shadow, a dynamic test and four numbers, it returns JVML instructions and a new environment. The first number represents the position in the code where to jump to if the test succeeds, which is the start of the advice. The second number is the position where to jump to if the test fails, which is the end of the advice. The third number is the number from which the function getTestInstructions starts numbering its instructions. The fourth number corresponds to the current maximum length of the method local variable table. The following notation is

used in the definition of getTestInstructions:

**Notation:** Given two maps $m$ and $m'$ with $\mathrm{Dom}(m) \cap \mathrm{Dom}(m') = \emptyset$, we write $m \dagger m'$ to denote the map where $\mathrm{Dom}(m \dagger m') = \mathrm{Dom}(m) \cup \mathrm{Dom}(m')$ and $(m \dagger m')(a) = m(a)$ if $a \in Dom(m)$ and $m'(a)$ otherwise.

The function getArgVar, used in getTestInstructions and described in Appendix II, allows the compiler to get the argument position in the method local variables table. Notice that we do not need a function that gets the position of the target because it is always at the highest position of the method local variables table. The environment may change because of the function genPool, listed in Appendix II, which may add new entries in the constant pool of the given method class. In fact, this function looks for a class in the constant pool of the method class. If the class is found, it returns the constant pool entry number describing this class. If the class is not found, it generates a new entry in the given constant pool and returns the new constant pool entry number.

The function lenTestCode is also given in Appendix II. It calculates the number of instructions to be generated for a given test.

**Example:**

The following example shows the bytecode generated by the instruction getTestInstructions($\mathcal{E}$,$m$,$s$,$t$,40,42,28) where:

- The dynamic test $t$ is `Or(And(this instanceof(`$A$`), arg(1) instanceof(`$B$`)), this instanceof(`$C$`))`.
- The shadow on which we want to apply the test is in a method $m$ of a class $X$.
- The position of the advice call is 40 which correspond to two consecutive JVML instructions(`invokevirtual` and `invokestatic`) [9].
- The position of the instruction following the advice call is 42.
- The position of the instruction from which getTestInstructions starts numbering

getTestInstructions: *Environment*×*Method*×*Shadow*×*Test*×*Nat*×*Nat*×*Nat* ×*Nat* → *Code*×*Environment*

getTestInstructions($\mathcal{E}$,*m*,*s*,And($t_1$,$t_2$),*yes*,*no*,*start*,*maxLocals*)=
getTestInstructions($\mathcal{E}$,*m*,*s*,$t_1$,*start*+lenTestCode($t_1$),*no*,*start*,*maxLoc*) †
  getTestInstructions($\mathcal{E}$,*m*,*s*,$t_2$,*yes*,*no*,*start*+lenTestCode($t_1$),*maxLoc*)

getTestInstructions($\mathcal{E}$,*m*,*s*,Or($t_1$,$t_2$),*yes*,*no*,*start*,*maxLoc*)=
  getTestInstructions($\mathcal{E}$,*m*,*s*,$t_1$,*yes*,*start*+lenTestCode($t_1$),*start*,*maxLoc*) †
  getTestInstructions($\mathcal{E}$,*m*,*s*,$t_2$,*yes*,*no*,*start*+lenTestCode($t_1$),*maxLoc*)

getTestInstructions($\mathcal{E}$,*m*,*s*,Not(*t*),*yes*,*no*,*start*,*maxLoc*)=
  getTestInstructions($\mathcal{E}$,*m*,*s*,*t*,*no*,*yes*,*start*,*maxLoc*)

getTestInstructions($\mathcal{E}$,*m*,*s*,`this instanceof (`*ct*`)`, *yes*, *no*, *start*, *maxLoc*) = (*c*, $\mathcal{E}'$)
where
  (*j*,$\mathcal{E}'$)=genPool($\mathcal{E}$,*m*,*ct*)
  *c*(*start*)= `aload_0`
  *c*(*start*+1)= `instanceof` *j*
  *c*(*start*+2)= `ifeq` *no*
  *c*(*start*+3)= `goto` *yes*

getTestInstructions($\mathcal{E}$,*m*,*s*,`target instanceof (`*ct*`)`, *yes*, *no*, *start*, *maxLoc*) = (*c*, $\mathcal{E}'$)
where
  (*j*,$\mathcal{E}'$)=genPool($\mathcal{E}$,*m*,*ct*)
  *c*(*start*)= `aload_`*maxLoc*
  *c*(*start*+1)= `instanceof` *j*
  *c*(*start*+2)= `ifeq` *no*
  *c*(*start*+3)= `goto` *yes*

getTestInstructions($\mathcal{E}$,*m*,*s*,`args (`*i*`) instanceof (`*ct*`)`, *yes*, *no*, *start*, *maxLoc*) = (*c*, $\mathcal{E}'$)
where
  (*j*,$\mathcal{E}'$)=genPool($\mathcal{E}$,*m*,*ct*)
  *c*(*start*)= `aload_`getArgVar(*m*,*s*,*maxLoc*,*i*)
  *c*(*start*+1)= `instanceof` *j*
  *c*(*start*+2)= `ifeq` *no*
  *c*(*start*+3)= `goto` *yes*

the JVML test instructions is 28. Previously, the instructions in $m$ have been renumbered in order to let free the positions from 28 to 40.

- A temporary variable has been created at the position 6 by the preparatory phase in order to contain the value of the first argument.
- The constant pool of $X$ contains at the entry 3 a description of the class $A$.
- The constant pool of $X$ contains at the entry 10 a description of the class $B$.
- The constant pool of $X$ doesn't initially contain an entry describing $C$. A new entry at position 35 is created in the constant pool for this effect.

The test instructions returned by getTestInstructions($\mathcal{E}$,$m$,$s$,$t$,40,42,28) are:

```
28: aload_0
29: instanceof #3
30: ifeq 36
31: goto 32
32: aload_6
33: instanceof #10
34: ifeq 36
35: goto 40
36: aload_0
37: instanceof #35
38: ifeq 42
39: goto 40
```

The code of $m$ is updated to $m.code$ † getTestInstructions($\mathcal{E}$,$m$,$s$,$t$,40,42,28). Notice that the code generated by getTestInstructions is not optimized. In fact, AspectJ optimizes the code by removing all the `goto` instructions that branch to the next instruction in the code. We easily check the code generated by getTestInstructions in order to

remove non-necessary instructions (31 and 39 in the example).

## 5.5 Weaving Semantics Rules

This section presents the semantic rules of the advice weaving in AspectJ. We first describe the semantic configurations in Table 5.5 with the type *Configuration*.

| | | |
|---|---|---|
| *Configuation* | ::= | *Environment*×*Method*× (*Shadow*) `-list` × (*Shadow*) `-list` ×*State*×*Nat* |
| *State* | ::= | `start` \| `shadowed` \| `matching` \| `startWeaving` \| `weaving` \| `weaved` |

Table 5.5: AspectJ Semantic Configuration

Configurations use a new type *State*, which represents the different stages of the advice weaving process: `start`, `shadowed`, `matching`, `startWeaving`,`weaving`, and `weaved`. A semantic configuration has the form: $\langle \mathcal{E}, m, sh, msh, state, maxLoc \rangle$ where $\mathcal{E}$ represents the environment, $m$ the current method, $sh$ the list of shadows in the method code, $msh$ the list of all shadows in $sh$ that match with one or more pointcuts present in the advices of the environment, $state$ the stage of the pointcut matching process and the $maxLoc$ the current maximum length of the method $m$ local variables table. We remind the reader that we need this value in order to store shadow arguments and target (if any) in temporary variables in the method local variables table in order to test them in case of advices with dynamic pointcuts. The weaving semantics starts with the configuration: $\langle \mathcal{E}, m, [\,], [\,], \texttt{start}, maxLoc \rangle$ and ends with the configuration $\langle \mathcal{E}, m, sh, msh, \texttt{shadowed}, maxLoc \rangle$ which represents the case where all the shadows contained in the method $m$ have been calculated and put in the list of shadows $sh$. The initial value of $maxLoc$ is the length of the method variables table. Once this step done, the configurations with the state `matching` will take all the shadows in $sh$ and try to match them with the pointcuts of all the advices in the environment $\mathcal{E}$. All the matched

shadows are then put in the list *msh*. In the stage `weaving`, we inject to each matched join point its corresponding advices that appear in the list *m*ungers of the shadow. The state `weaved` is reached once the complete weaving is done. The stage `startWeaving` is used to start the weaving for a given join point. It allows us to store the arguments and the target of the join points when one of its applicable advices has a dynamic pointcut. The entire semantics is specified in thirteen (13) rules, which employ utility functions described in previous sections or in Appendix II.

**Rule 1:** In this rule, the starting configuration has the state `start`. This rule allows the calculation of all the shadows in the method *m*. The utility function preShadowing allows us to calculate all the execution shadows whereas shadowing calculates the non-execution shadows. The environment and the method change because of the `impdep1` insertions. The mnemonics `impdep1` delimit the starting and end of all the possible shadows in the method *m*. We assume that all the methods have been rearranged to contain only one return statement:

$$\frac{\begin{array}{c}(\mathcal{E}_1, m_1, l) = \mathsf{preShadowing}(\mathcal{E}, m) \\ (\mathcal{E}_2, m_2, sh) = \mathsf{shadowing}(\mathcal{E}_1, m_1, 0, l)\end{array}}{\begin{array}{l}\langle \mathcal{E}, m, [\,], [\,], \texttt{start}, \mathsf{length}(m.methodVariables)\rangle \longrightarrow \\ \qquad \langle \mathcal{E}_2, m_2, sh, [\,], \texttt{shadowed}, \mathsf{length}(m.methodVariables)\rangle\end{array}}$$

**Rule 2:** This rule starts the pointcut matching process. It takes the first shadow in the shadow list *sh* and tries to match it with the pointcuts of the advices in the environment $\mathcal{E}$. If the result of the pointcut matching process is not empty, this shadow is added to the matched shadows list *msh* and the configuration state moves from `shadowed` to `matching`.

$$\frac{\begin{array}{c}shadow = \mathsf{match}(\mathcal{E}, m, \mathsf{head}(sh), \mathcal{E}.advices) \\ msh = \mathsf{ifThenElse}(shadow.mungers \neq [\,], [shadow], [\,])\end{array}}{\begin{array}{l}\langle \mathcal{E}, m, sh, [\,], \texttt{shadowed}, maxLoc\rangle \longrightarrow \\ \qquad \langle \mathcal{E}, m, \mathsf{tail}(sh), msh, \texttt{matching}, maxLoc\rangle\end{array}}$$

**Rule 3:** The third rule is very close to the second rule and corresponds to the case where there are still shadows to match against the pointcuts of the advices in $\mathcal{E}$. It treats the next shadow in the shadow list. The state is unchanged and remains `matching`.

$$\frac{\begin{array}{c} sh \neq [\,] \\ shadow = \mathsf{match}(\mathcal{E}, m, \mathsf{head}(sh), \mathcal{E}.advices) \\ msh_1 = \mathsf{ifThenElse}(shadow.mungers \neq [\,], msh :: shadow, msh) \end{array}}{\begin{array}{l} \langle \mathcal{E}, m, sh, msh, \texttt{matching}, maxLoc \rangle \longrightarrow \\ \qquad \langle \mathcal{E}, m, \mathsf{tail}(sh), msh_1, \texttt{matching}, maxLoc \rangle \end{array}}$$

**Rule 4:** This rule corresponds to the case where all the shadows have been considered for the pointcut matching process but where there is no advice weaving to apply because none of the program shadows match any of the advice pointcuts. The process of advice weaving is then stopped and the configuration state changes from `matching` to `weaved`.

$$\frac{\begin{array}{c} sh = [\,] \\ msh = [\,] \end{array}}{\begin{array}{l} \langle \mathcal{E}, m, sh, msh, \texttt{matching}, maxLoc \rangle \longrightarrow \\ \qquad \langle \mathcal{E}, m, sh, msh, \texttt{weaved}, maxLoc \rangle \end{array}}$$

**Rule 5:** This rule depicts the case where all the shadows have been considered for the pointcut matching process and the set of matched shadows *msh* is not empty and advice weaving applies. In this case the configuration state changes from `matching` to `startWeaving`.

$$\frac{\begin{array}{c} sh = [\,] \\ msh \neq [\,] \end{array}}{\begin{array}{l} \langle \mathcal{E}, m, sh, msh, \texttt{matching}, maxLoc \rangle \longrightarrow \\ \qquad \langle \mathcal{E}, m, sh, msh, \texttt{startWeaving}, maxLoc \rangle \end{array}}$$

**Rule 6:** We consider in this rule the case where one of the current join point's applicable advices has a dynamic pointcut. In this case, we must inject the instructions that store all the arguments and the target of the current shadow in temporary variables before starting to inject any advice. We store the arguments and the target in order to test then when necessary. Notice that we reload the arguments and the target onto the stack in case where they were already there:

$$\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \mathsf{oneIsDynamic}(\mathsf{head}(msh).mungers) \\ il = \mathsf{storeCode}(m, \mathsf{head}(msh), \mathsf{argumentTypes}(\mathsf{head}(msh)), maxLoc) \\ (\mathcal{E}_1, m_1, msh_1) = \mathsf{liberateAt}(\mathcal{E}, m, msh, \mathsf{length}(il), \mathsf{head}(msh).start) \\ (\mathcal{E}_2, m_2) = \mathsf{insertStore}(\mathcal{E}_1, m_1, msh_1, il) \\ target = \mathsf{ifThenElse}(\mathsf{hasTarget}(\mathsf{head}(msh_1), m_2), 1, 0) \\ il_1 = \mathsf{ifThenElse}(\pi 2(\mathsf{head}(msh).kind) = \texttt{true}, \mathsf{loadCode}(il), []) \\ maxLoc_1 = maxLoc + \mathsf{length}(\mathsf{argumentTypes}(\mathsf{head}(msh_1))) + target \end{array}}{\begin{array}{c} \langle \mathcal{E}, m, [\,], msh, \texttt{startWeaving}, maxLoc \rangle \longrightarrow \\ \langle \mathcal{E}_2, m_2, [\,], msh_1, \texttt{weaving}, maxLoc_1 \rangle \end{array}}$$

**Rule 7:** We consider in this rule the case where none of the current join point's applicable advices has a dynamic pointcut. In this case, we do not inject the instructions that store all the arguments and the target of the current shadow in temporary variables before starting to inject the advices. The state changes from `startWeaving` to `weaving`:

$$\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \neg\mathsf{oneIsDynamic}(\mathsf{head}(msh).mungers) \end{array}}{\begin{array}{c} \langle \mathcal{E}, m, [\,], msh, \texttt{startWeaving}, maxLoc \rangle \longrightarrow \\ \langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc \rangle \end{array}}$$

**Rule 8:** In this rule, the list of mungers of the current head of the matched shadows list is not empty yet and needs to be weaved to the shadow. The head of this list of mungers is a munger with a test equal to `always`. In this case, we need only inject the instructions representing the advice call using the utility function insertAdvice. The environment, the method, and the matched shadows change because of the insertion of new JVML instructions (representing the advice call) inside the method *m*. More precisely, each matched shadow *s* may have to readjust the values of its parts *start* and *end* if the shadow is affected by the advice call insertion. These updates appear clearly in the method liberate, which allows us to free places in the code in order to inject the new instructions:

$$\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest = \texttt{always} \\ (\mathcal{E}_1, m_1, msh_1) = \mathsf{liberate}(\mathcal{E}, m, msh, 2) \\ (\mathcal{E}_2, m_2) = \mathsf{insertAdvice}(\mathcal{E}_1, m_1, msh_1) \\ msh_2 = \mathsf{head}(msh_1)[mungers \leftarrow \mathsf{tail}(\mathsf{head}(msh_1).mungers)] :: tail(msh_1) \end{array}}{\begin{array}{l} \langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc \rangle \longrightarrow \\ \qquad \langle \mathcal{E}_2, m_2, [\,], msh_2, \texttt{weaving}, maxLoc \rangle \end{array}}$$

**Rule 9:** We consider in this rule the case where the advice is a before advice but where its pointcut is dynamic. In this case, we must inject the instructions corresponding to the dynamic test followed by the advice call:

$$\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest \neq \texttt{always} \\ \mathsf{head}(\mathsf{head}(msh).mungers).adviceInfo.akind = \texttt{Before} \\ (\mathcal{E}_1, m_1, msh_1) = \mathsf{liberateAt}(\mathcal{E}, m, msh, 2, \mathsf{head}(msh).start) \\ (\mathcal{E}_2, m_2) = \mathsf{insertBeforeAdvice}(\mathcal{E}_1, m_1, msh_1) \\ lenTest = \mathsf{lenTestCode}(\mathsf{head}(\mathsf{head}(msh_1).mungers).pointcutTest \\ (\mathcal{E}_3, m_3, msh_2) = \mathsf{liberateAt}(\mathcal{E}_2, m_2, msh_1, lenTest, \mathsf{head}(msh).start) \\ (\mathcal{E}_4, m_4) = \mathsf{insertTestBeforeInstructions}(\mathcal{E}_3, m_3, msh_2, maxLoc) \\ msh_3 = \mathsf{head}(msh_2)[mungers \leftarrow \mathsf{tail}(\mathsf{head}(msh_2).mungers)] :: tail(msh_2) \end{array}}{\begin{array}{l} \langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc \rangle \longrightarrow \\ \qquad \langle \mathcal{E}_4, m_4, [\,], msh_3, \texttt{weaving}, maxLoc \rangle \end{array}}$$

**Rule 10:**

The following rule is similar to the previous rule except that the advice is an after advice. More precisely, it corresponds to the case where the shadow is not an execution shadow:

$$
\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest \neq \texttt{always} \\ \mathsf{head}(\mathsf{head}(msh).mungers).adviceInfo.akind = \texttt{After} \\ \pi 2(\mathsf{head}(msh).kind) = \texttt{true}//\textit{not execution shadow} \\ (\mathcal{E}_1, m_1, msh_1) = \mathsf{liberateAt}(\mathcal{E}, m, msh, 2, \mathsf{head}(msh).end) \\ (\mathcal{E}_2, m_2) = \mathsf{insertAfterAdvice}(\mathcal{E}_1, m_1, msh_1) \\ lenTest = \mathsf{lenTestCode}(\mathsf{head}(\mathsf{head}(msh_1).mungers).pointcutTest \\ (\mathcal{E}_3, m_3, msh_2) = \mathsf{liberateAt}(\mathcal{E}_2, m_2, msh_1, lenTest, \mathsf{head}(msh).end) \\ (\mathcal{E}_4, m_4) = \mathsf{insertTestAfterInstructions}(\mathcal{E}_3, m_3, msh_2, maxLoc) \\ msh_3 = \mathsf{head}(msh_2)[mungers \leftarrow \mathsf{tail}(\mathsf{head}(msh_2).mungers)] :: tail(msh_2) \end{array}}{\begin{array}{c} \langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc \rangle \longrightarrow \\ \langle \mathcal{E}_4, m_4, [\,], msh_3, \texttt{weaving}, maxLoc \rangle \end{array}}
$$

**Rule 11:**

This rule also depicts the case where the advice is an after advice except that here the shadow is an execution shadow. In this case, the advice instructions are injected just before the return instruction of the method:

$$
\frac{\begin{array}{c} msh \neq [\,] \\ \mathsf{head}(msh).mungers \neq [\,] \\ \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest \neq \texttt{always} \\ \mathsf{head}(\mathsf{head}(msh).mungers).adviceInfo.akind = \texttt{After} \\ \pi 2(\mathsf{head}(msh).kind) = \texttt{false}//\textit{execution shadow} \\ lenTest = \mathsf{lenTestCode}(\mathsf{head}(\mathsf{head}(msh_1).mungers).pointcutTest \\ (\mathcal{E}_3, m_3, msh_2) = \mathsf{liberateAt}(\mathcal{E}_2, m_2, msh_1, lenTest{+}2, \mathsf{head}(msh_1).end{-}2) \\ (\mathcal{E}_4, m_4) = \mathsf{insertTestAfterInstructions}(\mathcal{E}_3, m_3, msh_2, maxLoc) \\ (\mathcal{E}_5, m_5) = \mathsf{insertAfterAdvice}(\mathcal{E}_4, m_4, msh_2) \\ msh_3 = \mathsf{head}(msh_2)[mungers \leftarrow \mathsf{tail}(\mathsf{head}(msh_2).mungers)] :: tail(msh_2) \end{array}}{\begin{array}{c} \langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc \rangle \longrightarrow \\ \langle \mathcal{E}_5, m_5, [\,], msh_3, \texttt{storeweaving}, maxLoc \rangle \end{array}}
$$

**Rule 12:** This rule depicts the case where all the possible advices have been already injected to the shadow

in the head of the matched shadows list. This latter is then removed from the list and the weaving process is restarted with the next join point if any:

$$\frac{msh \neq [\,] \qquad \mathsf{head}(msh).mungers = [\,]}{\langle \mathcal{E}, m, [\,], msh, \texttt{weaving}, maxLoc\rangle \longrightarrow \langle \mathcal{E}, m, [\,], \mathsf{tail}(msh), \texttt{startWeaving}, maxLoc\rangle}$$

**Rule 13:** This rule corresponds to the case where all the shadows have been considered for the weaving process. The process of advice weaving is then stopped and the configuration state changes from `weaving` to `weaved`.

$$\frac{msh = [\,]}{\langle \mathcal{E}, m, sh, msh, \texttt{startWeaving}, maxLoc\rangle \longrightarrow \langle \mathcal{E}, m, [\,], msh, \texttt{weaved}, maxLoc\rangle}$$

## 5.6 Conclusion

In this chapter, we reported a formalization of the advice weaving in AspectJ. The chapter clarifies and helps to understand in depth the matching and advice injection processes in AspectJ, especially in case of dynamic pointcuts. We formalized the residues generation in AspectJ and we described how the dynamic tests are generated in the case of dynamic pointcuts. Such a formalization allows us to easily notice, for example, that a "target" pointcut never matches a constructor call join point or that an "args" pointcut never matches a field get join point, which may not be obvious for new AspectJ users. Hence, the semantics presented in this chapter can be of considerable help in understanding the inner workings of AspectJ compilers.

In the next chapter, we will describe another advice weaving semantics but for an AOP calculus based on λ-calculus. This is principally motivated by the desire to remove the syntactic dissimilarities between any related constructs in different AOP languages. Furthermore, this calculus contains the data flow pointcut that is very useful from security standpoint.

# Chapter 6

# AOP Security Calculus

This chapter presents an aspect oriented calculus for security that is based on λ-calculus and called λ-SAOP. It is an AOP extension of the extended lambda calculus presented in Section 2.4.4 and contains pointcuts that are relevant to application security hardening. The main contribution of the chapter is a semantics for λ-SAOP advice weaving. Section 6.1 describes the syntax of the language. A detailed description of the types and the tags of λ_SAOP is given in Section 6.2. The matching and the weaving processes are presented in Section 6.3 . Finally, Section 6.4 shows how the type inference algorithm is accommodated to take the matching and the weaving processes into consideration.

## 6.1 Syntax

The main specificities of λ_SAOP are:

- λ_SAOP deals the following pointcuts: `call`, `get`, `set`, and `dflow`. These pointcuts are useful from security perspective. The `call` pointcut picks out join points where functions are called. The `set` and `get` pointcuts pick out join points where variables are set and read respectively. The `dflow` pointcut identifies join points based on the origins of data. These pointcuts are important to inject security code at specific points.
- The weaving in λ-SAOP is in the spirit of AspectJ where advices are injected before, after, or around the join points that match their respective pointcuts. We use the sequence construct Ş;Ť of the extended λ-calculus to perform the injection.

- In λ_SAOP, data dependencies between expressions are statically tracked using data flow tags.
- λ_SAOP uses an effect-based type system to infer types.
- The effect-based type inference algorithm is accommodated to take the matching and the weaving processes into consideration.

A λ_SAOP program, as shown in Figure 6.1 and Figure 6.2, consists of an expression and a sequence of advices. These expressions represent the extended lambda calculus presented in Section 2.4.4 and can belong to one of the following categories:

- Constants and variables.
- Functional constructs such as function abstraction, function application, and recursion.
- Let expressions.
- Sequencing.
- Imperative notations such as referencing. An expression of the form **ref**($e$) allows the allocation of a new reference that points to the value obtained from the evaluation of $e$. The unary operator " ! " is used for dereferencing, and the binary operator " := " is used for assignment.

Each advice has a kind (*akind*) that can be either `before`, `after`, or `around`. It contains also a pointcut designator (*pcd*) that specifies the join points in which it is interested, and a body (*exp*) representing the action to be taken at those points. In the case of `around` advice, the body (*exp*) may contain a special variable *proceed* that represents the advice with next precedence, or the computation under the join point if there is no further advice. *AdvSeq* is a sequence of advices. Empty sequence is represented by the symbol ε. We consider four kinds of basic pointcuts: `call`, `set`, `get`, and `dflow`. The pointcut syntax uses type schemes to specify join point types and tags to discriminate data flow pointcuts. Types and tags are detailed in the next section. Basic pointcuts can be logically combined to produce more complex ones using boolean operators. *Vname* is an infinite set of variable names whereas *Fname* is an infinite set of function names. Notice that $Fname \cap Vname = \emptyset$. An integer value is represented by $\mathbf{n}$.

## 6.2 Types and Tags

Data flow pointcuts are very interesting pointcuts for application security hardening as mentioned in Section 3.2.2. Masuhara and Kawauchi [66] have defined the data flow pointcut for security purposes using data

| | | | | | |
|---|---|---|---|---|---|
| *Prog* | ∋ | *p* | ::= | $e \triangleleft s$ | **(Program)** |
| *Exp* | ∋ | *e* | ::= | *c* | **(Expressions)** |
| | | | \| | *x* | |
| | | | \| | $\lambda x.e$ | |
| | | | \| | $e_1 e_2$ | |
| | | | \| | **let rec** f x = $e_1$ **in** $e_2$ | |
| | | | \| | **let** x = $e_1$ **in** $e_2$ | |
| | | | \| | $e_1;e_2$ | |
| | | | \| | **ref** *e* | |
| | | | \| | **!** *e* | |
| *Const* | ∋ | *c* | ::= | **n** \| **( )** \| **true** \| **false** | **(Constants)** |
| *AdvSeq* | ∋ | *s* | ::= | ⟨*akind*: `before` \| `after`, | **(Advices)** |
| | | | | *pcd*: *Pcd*, | |
| | | | | *exp*: *Exp* ⟩ *s* | |
| | | | \| | ⟨*akind*: `around`, | |
| | | | | *pcd*: *Pcd*, | |
| | | | | *exp*: *ExpAr* ⟩ *s* \| ε | |
| *ExpAr* | ∋ | $e'$ | ::= | *c* | **(Around Expressions)** |
| | | | \| | *x* | |
| | | | \| | *proceed* | |
| | | | \| | $\lambda x.e'$ | |
| | | | \| | $e'_1 e'_2$ | |
| | | | \| | **let rec** f x = $e'_1$ **in** $e'_2$ | |
| | | | \| | **let** x = $e'_1$ **in** $e'_2$ | |
| | | | \| | $e'_1;e'_2$ | |
| | | | \| | **ref** $e'$ | |
| | | | \| | **!** $e'$ | |
| | | | \| | $e'_1$ := $e'_2$ | |
| *Pcd* | ∋ | *p* | ::= | *true* \| ¬*p* \| *p* ∧*p* \| *cp* \| *sgp* \| *dp* | **(Pointcuts )** |

Figure 6.1: λ_SAOP Syntax Part I

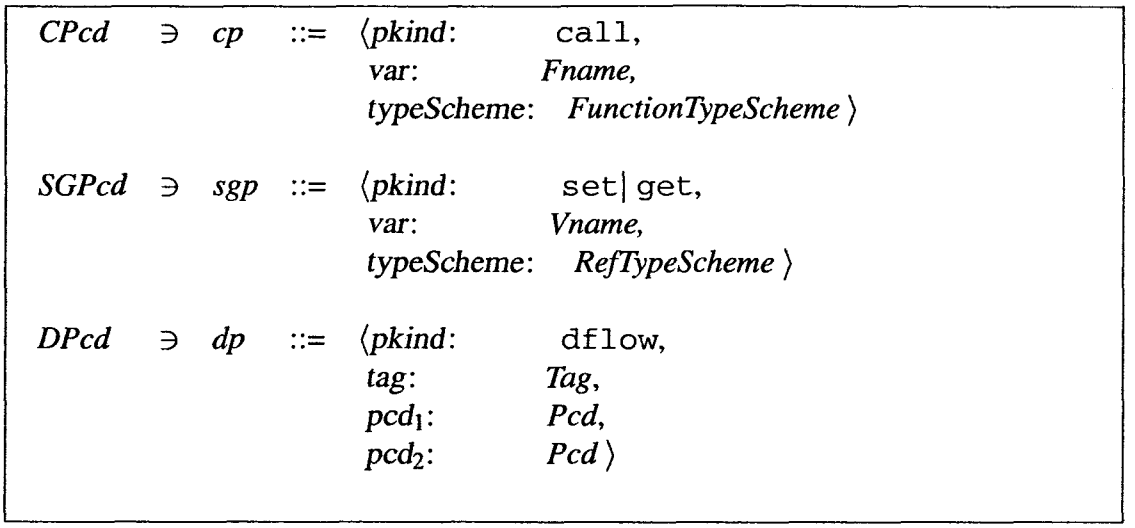



Figure 6.2: λ_SAOP Syntax Part II

flow tags. However, they have not provided a formal framework for this pointcut and in their approach, data flow tags are propagated and inspected at run-time. The data flow pointcut `dflow[x,x'](p)` as defined by Masuhara and Kawauchi matches if there is a data flow from `x'` to `x`. Variable `x` should be bound to a value in the current join point whereas variable `x'` should be bound to a value in a past join point matching to `p`. Therefore, Masuhara and Kawauchi's data flow pointcut must be used in conjunction with some other pointcut that binds `x` to a value in the current join point.

In this thesis, a formal static framework is defined to match and weave data flow pointcuts. Data flow tags are propagated statically to track data dependencies between expressions during typing. This reduces the run-time checking overhead considerably. In addition, the defined data flow pointcut can be used without being conjuncted with some other pointcuts. Hence, it can be used alone and if needed it can be conjuncted with other pointcuts. This makes it more similar to the other known pointcuts. Figure 6.3 formally presents the types and the tags in λ-SAOP. The considered types are: *int*, *bool*, *unit*, functional types $\tau \xrightarrow{\eta} \tau'$ (where $\tau$ and $\tau'$ are types too), reference types $ref_\rho(\tau)$, and type variables. We also define type schemes of the form $\forall \upsilon_1 ... \upsilon_n.\tau$ where $\upsilon$ can be a type, a region, or an effect variable. Two special type schemes are defined: *FunctionTypeScheme* and *RefTypeScheme*. *FunctionTypeScheme* is used with `call` pointcut while *RefTypeScheme* is used with `set` and `get` pointcuts. Tags (*Tag*) as previously mentioned are used to discriminate data flow pointcuts and are represented by natural numbers. Effects can be gathered together with the infix ";" that denotes the union of effects.

The data flow pointcut, as shown in Figure 6.2, has two pointcuts as parts of its syntax and a tag that discriminates this pointcut from the other defined data flow pointcuts. If an expression matches the first pointcut of a datflow pointcut, this expression is tagged with the tag of this data flow pointcut. This

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *Region* | $\ni$ | $\rho$ | ::= | $r$ | \| | $\gamma$ | | |
| *Effect* | $\ni$ | $\eta$ | ::= | $\emptyset$ | \| | $\varsigma$ | \| | $\eta;\eta$ |
| | | | \| | $init(\rho,\tau)$ | \| | $read(\rho,\tau)$ | \| | $write(\rho,\tau)$ |
| *Type* | $\ni$ | $\tau$ | ::= | *int* | \| | *bool* | \| | *unit* |
| | | | \| | $\tau \xrightarrow{\eta} \tau$ | \| | $ref_\rho(\tau)$ | \| | $\alpha$ |
| *TypeScheme* | $\ni$ | $\sigma$ | ::= | $\tau$ | \| | $\forall\upsilon\sigma$ | | |
| *FunctionTypeScheme* | $\ni$ | $\phi$ | ::= | $\tau \xrightarrow{\eta} \tau$ | \| | $\forall\upsilon\phi$ | | |
| *RefTypeScheme* | $\ni$ | $\varphi$ | ::= | $ref_\rho(\tau)$ | \| | $\forall\upsilon\varphi$ | | |
| *Tag* | $\ni$ | $\iota$ | ::= | $\mathbf{n}$ | | | | |

Figure 6.3: Types and Tags

tag is then propagated to other expressions that are data-dependent on the expression that matches the first pointcut. Finally, if an expression matches the second pointcut of this data flow pointcut and is tagged with the tag of this data flow pointcut which means that it depends on the the first expression that matches the first pointcut, then it matches the corresponding data flow pointcut. On the other hand, if an expression matches the second pointcut of this data flow pointcut and it is not tagged with the tag of this data flow pointcut which means that it does not depend on the first expression that matches the first pointcut, it will not match the corresponding data flow pointcut. The maximum number of tags in a set associated with an expression is equal to the number of the defined data flow pointcuts.

The set of tags associated with an expression is specified according to the tagging rules specified in Figure 6.4.

The data flow judgment $\mathcal{E},s,m \vdash_d e : t,m'$ is used to specify that an expression $e$ is associated with a set of tags $t$ in the presence of a sequence of defined advices $s$ where a tagging environment $\mathcal{E}$ maps variables to tag sets. The concept of a tagging environment $\mathcal{E}$ is similar to the concept of a typing environment $\Gamma$ and at the same time the domains of both environments are equal. A mapping $m$ stores mappings from regions to tag sets. The mapping $m'$ reflects a modified version of $m$ after tagging the expression $e$. We assume that expressions are $\alpha$-converted. The function M, presented in Appendix III, checks if the type is a reference type in region $\rho$ and if $\rho$ is associated with the tag set $t$ in the mapping $m$. If so it returns $t$. Otherwise, it returns an empty set. Comparing types depends on pattern matching and not on type unification because the later changes both types so they become equal while in this case all what we need is to check if the type is a reference type or not. The functions searchTagCall, searchTagGet, and searchTagSet work on a sequence of advices and return a set of tags. The tags returned from the first one discriminate the data flow pointcuts that their first pointcuts match a specific application expression, the tags returned from the second one discriminate the data flow pointcuts that their first pointcuts match a specific dereferencing expression,

$$\mathcal{E},s,m \vdash_d c : \{\},m \qquad \textbf{(Const)}$$

$$\frac{x:t \in \mathcal{E} \quad \Gamma \vdash x:\tau,\eta}{\mathcal{E},s,m \vdash_d x : \mathsf{M}(\tau,m) \cup t,m} \qquad \textbf{(Var)}$$

$$\frac{\mathcal{E}_x\dagger[x \mapsto \{\}],s,m \vdash_d e : t,m'}{\mathcal{E},s,m \vdash_d \lambda x.e : t,m'} \qquad \textbf{(Abs)}$$

$$\frac{\mathcal{E},s,m \vdash_d e_1 : t_1,m' \quad \mathcal{E},s,m' \vdash_d e_2 : t_2,m'' \quad \Gamma \vdash e_1 : \tau,\eta \quad \Gamma \vdash e_1 e_2 : \tau',\eta'}{\mathcal{E},s,m \vdash_d e_1 e_2 : \mathsf{M}(\tau',m'') \cup t_1 \cup t_2 \cup \mathsf{searchTagCall}(e_1,\tau,t_2,s),m''} \qquad \textbf{(App)}$$

$$\frac{\mathcal{E},s,m \vdash_d e_1 : t_1,m' \quad \mathcal{E},s,m' \vdash_d e_2 : t_2,m''}{\mathcal{E},s,m \vdash_d e_1;e_2 : t_2,m''} \qquad \textbf{(Seq)}$$

$$\frac{\mathcal{E}_{x,f}\dagger[x \mapsto \{\},f \mapsto \{\}],s,m \vdash_d e_1 : t_1,m' \quad \mathcal{E}_f\dagger[f \mapsto t_1],s,m' \vdash_d e_2 : t_2,m''}{\mathcal{E},s,m \vdash_d \textbf{let rec } f\ x = e_1 \textbf{ in } e_2 : t_2,m''} \qquad \textbf{(Letrec)}$$

$$\frac{\mathcal{E},s,m \vdash_d e_1 : t_1,m' \qquad \mathcal{E}_x\dagger[x \mapsto t_1],s,m' \vdash_d e_2 : t_2,m''}{\mathcal{E},s,m \vdash_d \textbf{let } x = e_1 \textbf{ in } e_2 : t_2,m''} \qquad \textbf{(Let)}$$

$$\frac{\mathcal{E},s,m \vdash_d e : t,m'}{\mathcal{E},s,m \vdash_d \textbf{ref } (e) : t,m'} \qquad \textbf{(Ref)}$$

$$\frac{\mathcal{E},s,m \vdash_d e : t,m' \quad \Gamma \vdash e : \tau,\eta \quad \Gamma \vdash !e : \tau',\eta'}{\mathcal{E},s,m \vdash_d !e : \mathsf{M}(\tau',m') \cup t \cup \mathsf{searchTagGet}(e,\tau,t,s),m'} \qquad \textbf{(Deref)}$$

$$\frac{\mathcal{E},s,m \vdash_d e_1 : t_1,m' \quad \mathcal{E},s,m' \vdash_d e_2 : t_2,m'' \quad \Gamma \vdash e_1 : ref_\rho(\tau),\eta}{\mathcal{E},s,m \vdash_d e_1 := e_2 : t,m''_\rho\dagger[\rho \mapsto t_2]} \qquad \textbf{(Assign)}$$

where $t = t_1 \cup t_2 \cup \mathsf{searchTagSet}(e_1, ref_\rho(\tau), t_2, s)$

Figure 6.4: Tagging Rules

and the tags returned from the third one discriminate the data flow pointcuts that their first pointcuts match a specific assignment expression.

We refer the reader to Appendix III to understand the formal definitions of the utility functions that are used in this chapter.

An expression $e$ matches the `dflow` pointcut $\langle pkind{:}\texttt{dflow}, tag{:}\mathbf{n}, pcd_1{:}p_1, pcd_2{:}p_2\rangle$ if $e$ matches $p_2$, $e$ is data-dependent on a previous expression $e'$, and $e'$ matches $p_1$. To track data dependencies between expressions, all the expressions that match $p_1$, $e'$ is one of them, will be tagged with the tag of the data flow pointcut $\mathbf{n}$ and $\mathbf{n}$ will then be transmitted according to the defined tagging rules to other expressions, $e$ is one of them, because that are data-dependent on $e'$ . Accordingly, if $e$ is an application expression, $e$ matches the `dflow` pointcut $\langle pkind{:}\texttt{dflow}, tag{:}\mathbf{n}, pcd_1{:}p_1, pcd_2{:}p_2\rangle$ if it matches $p_2$ and its argument is tagged with $\mathbf{n}$. If $e$ is an assignment expression, it matches the pointcut if it matches $p_2$ and the right-hand side of the assignment operator is tagged with $\mathbf{n}$. If $e$ is a dereferencing expression, it matches the pointcut if it matches $p_2$ and the dereferencing argument is tagged with $\mathbf{n}$. This means that these expressions are data-dependent on a previous expression that matches the first pointcut $p_1$ of this data flow pointcut . The following example demonstrates the basic ideas related to the data flow pointcut:

$$
\begin{array}{l}
\textbf{let } x = \textbf{ref } 3 \\
\quad \textbf{in let } y = \textbf{ref } 4 \\
\quad\quad \textbf{in let } f = \lambda x.x \\
\quad\quad\quad \textbf{in } x{:=}!y; f(x)
\end{array}
$$

The `dflow` pointcut $\langle pkind{:}\texttt{dflow}, tag{:}k, pcd_1{:}p_1, pcd_2{:}p_2\rangle$ is matched by the expression $f(x)$ where the pointcut $p_1$ picks out join points where we dereference a variable $y$ of type $\forall\alpha\rho.ref_\rho(\alpha)$ and the pointcut $p_2$ picks out join points where we call a function $f$ of type $\forall\alpha\eta.\alpha \xrightarrow{\eta} \alpha$. This is justified by the following:

- The expression $!y$ satisfies the pointcut $p_1$ and consequently $!y$ is tagged with the tag $\mathbf{k}$ of the data flow pointcut according to (**Deref**) tagging rule.
- The assignment expression $x{:=}!y$ is then tagged with $\mathbf{k}$ according to the (**Assign**) tagging rule.
- $x$ depends on dereferencing $y$. Accordingly, the region $\rho$ of $x$ is associated with the tag $\mathbf{k}$ and stored in the mapping $m$ as indicated in the (**Assign**) tagging rule. Afterward, the tag associated with the region $\rho$ of $x$ will be retrieved from the mapping $m$ wherever $x$ appears using the function M. Hence, $x$ in $f(x)$ will be tagged with $\mathbf{k}$ according to the (**Var**) tagging rule.
- Finally, we conclude that $f(x)$ matches the defined pointcut because it satisfies the pointcut $p_2$ and $x$ is tagged with $\mathbf{k}$.

Now let us turn to the explanation of the tagging rules of Figure 6.4. Constants are associated with empty sets of tags. The tagging of variables is dictated by the tagging environment. Besides, we must take into consideration if the variable has a reference type using the function M. The tags associated with a function abstraction depend on the tags that are associated with its subexpression. For an application expression, its tag set contains the tags associated with the function expression and the tags associated with the argument. Besides, we must take into consideration wether the application expression has a reference type using the function M. Moreover, it contains the tags of the data flow pointcuts that are retrieved using the function (searchTagCall). The first pointcuts of these data flow pointcuts match the corresponding application expression. The tags associated with a sequence expression depend on the tags that are associated with its second subexpression. For a recursive let expression, the tags associated with it depend on the tags that are associated with its second subexpression provided that we extend the tagging environment with variable assumption for the function name. The tags associated with the function name are set to the tags that are associated with the first subexpression provided that we extend the tagging environment with variable assumptions. A similar explanation applies to the tagging of let expression where the tags associated with it depend on the tags that are associated with its second subexpression provided that we extend the tagging environment with variable assumption. The tags associated with a reference expression depend on the tags of its subexpression. For a dereferencing expression, its tag set contains the tags that are associated with its subexpression. Besides, we must take into consideration if the dereferencing expression has a reference type using the function M. Moreover, it contains the tags of the data flow pointcuts that are retrieved using the function (searchTagGet). The first pointcuts of these data flow pointcuts match the corresponding dereferencing expression. For an assignment expression, its tag set contains the tags that are associated with its first subexpression and the tags that are associated with its second subexpression. Moreover, it contains the tags of the data flow pointcuts that are retrieved using the function (searchTagSet). The first pointcuts of these data flow pointcuts match the corresponding assignment expression. After tagging the assignment expression, to keep the fact that the subexpression on the left-hand side of the assignment operator depends on the subexpression on the right-hand side of the assignment operator, the mapping $m$ is changed to reflect that the region of the subexpression on the left-hand side of the assignment operator is associated with the tag set of the subexpression on the right-hand side of the assignment operator. This is done to use the resulting set with this subexpression if it is used elsewhere and consequently to maintain the data-dependency between expressions.

The tagging algorithm $\mathcal{TG}$ regarding the tagging rules is detailed in Figure 6.5. It takes a tagging environment $\mathcal{E}$, a typing environment $\Gamma$, a sequence of defined advices $s$, a mapping $m$, and an expression $e$. It returns a set of tags that tags the expression $e$ and a modified version of $m$ after tagging the expression $e$. The algorithm $\mathcal{M}$ checks if the type is a reference type in a region variable $\gamma$ and if $\gamma$ is associated with

the tag set $t$ in $m$. If so it returns $t$. Otherwise, it returns an empty set. Checking if the variable is a reference type depends on pattern matching.

The algorithm $\mathcal{M}$ is given hereafter.

$\mathcal{M}(\tau,m)$= **case** $\tau$ **of**

$ref_\gamma(\tau')$ $\Rightarrow m(\gamma)$

**else** $\Rightarrow\{\}$

In the following, we state and prove a result that establishes the soundness of the tagging algorithm given of Figure 6.5. This proof is needed to prove the soundness of the inference algorithm for the type-based weaving rules as we will see next.

**Theorem: Soundness**

Given a tagging environment $\mathcal{E}$, a typing environment $\Gamma$, a sequence of defined advices $s$, a mapping $m$, and an expression $e$. If $\mathcal{TG}(\mathcal{E},\Gamma,s,m,e) = (t,m')$ then $\mathcal{E},s,m \vdash_d e : t,m'$.

**Proof**

The proof is done by structural induction on the expression:

- **Case of (Var)**

  Whenever $(t,m) = \mathcal{TG}(\mathcal{E},\Gamma,s,m,x)$ then by the definition of the algorithm

  $$[x \mapsto t'] \in \mathcal{E} \text{ where } t' \subseteq t.$$
  $$(\theta,\tau,\eta,k) = I(\Gamma,x)$$

  By the soundness proof of the effect-based inference algorithm $I$ [89]:

  $$\bar{k}\theta\Gamma \vdash x : \bar{k}\tau,\bar{k}\eta$$

  By the definition of the rule **(Var)**:

  $$\mathcal{E},s,m \vdash_d x : \mathsf{M}(\bar{k}\tau,m) \cup t',m$$

- **Case of (Abs)**

  By the definition of the algorithm: $\mathcal{TG}(\mathcal{E},\Gamma,s,m,\lambda x.e) = \mathcal{TG}(\mathcal{E}_x\dagger[x \mapsto \{\}],\Gamma,s,m,e)$

  Assume that $(t,m') = \mathcal{TG}(\mathcal{E}_x\dagger[x \mapsto \{\}],\Gamma,s,m,e)$

  By induction hypothesis on $e$:

  $$\mathcal{E}_x\dagger[x \mapsto \{\}],s,m \vdash_d e : t,m'$$

  By the definition of the rule **(Abs)**:

$\mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$ = **case** $e$ **of**

| | |
|---|---|
| $c$ | $\Rightarrow (\{\},m)$ |
| $x$ | $\Rightarrow$ **if** $x \notin \mathrm{Dom}(\mathcal{E})$ **then** *fail*<br>**else**<br>**let** $(\theta,\tau,\eta,k) = I(\Gamma,x)$<br>**in**<br>$(\mathcal{E}(x) \cup \mathcal{M}(\bar{k}\tau,m),m)$ |
| $\lambda x.e$ | $\Rightarrow \mathcal{TG}(\mathcal{E}_x\dagger[x \mapsto \{\}],\Gamma,s,m,e)$ |
| $e_1 e_2$ | $\Rightarrow$ **let** $(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$<br>$(t_2,m'') = \mathcal{TG}(\mathcal{E},\Gamma,s,m',e_2)$<br>$(\theta_1,\tau_1,\eta_1,k_1) = I(\Gamma,e_1)$<br>$(\theta_2,\tau_2,\eta_2,k_2) = I(\Gamma,e_1 e_2)$<br>**in**<br>$(t_1 \cup t_2 \cup \mathcal{M}(\bar{k}_2\tau_2,m'') \cup \mathsf{searchTagCall}(e_1,\bar{k}_1\tau_1,t_2,s),m'')$ |
| **let** $x = e_1$ **in** $e_2$ | $\Rightarrow$ **let** $(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$<br>$(t_2,m'') = \mathcal{TG}(\mathcal{E}_x\dagger[x \mapsto t_1],\Gamma,s,m',e_2)$<br>**in**<br>$(t_2,m'')$ |
| **ref** $e$ | $\Rightarrow \mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$ |
| $!e$ | $\Rightarrow$ **let** $(t,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$<br>$(\theta_1,\tau_1,\eta_1,k_1) = I(\Gamma,e)$<br>$(\theta_2,\tau_2,\eta_2,k_3) = I(\Gamma,!e)$<br>**in**<br>$(t \cup \mathcal{M}(\bar{k}_2\tau_2,m') \cup \mathsf{searchTagGet}(e,\bar{k}_1\tau_1,t,s),m')$ |
| $e_1 := e_2$ | $\Rightarrow$ **let** $(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$<br>$(t_2,m'') = \mathcal{TG}(\mathcal{E},\Gamma,s,m',e_2)$<br>$(\theta,\tau,\eta,k) = I(\Gamma,e_1)$<br>**in**<br>**if** $\bar{k}\tau = ref_\gamma(\tau')$ **then**<br>$(t_1 \cup t_2 \cup \mathsf{searchTagSet}(e_1,\bar{k}\tau,t_2,s),m''_\gamma\dagger[\gamma \mapsto t_2])$<br>**else** *fail* |

Figure 6.5: Tagging Algorithm

$$\mathcal{E},s,m \vdash_d \lambda x.e : t,m'$$

- **Case of (App)**

  By the definition of the algorithm:

$$(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$$
$$(t_2,m'')\mathcal{TG}(\mathcal{E},\Gamma,s,m',e_2)$$
$$(\theta_1,\tau_1,\eta_1,k_1) = I(\Gamma,e_1)$$
$$(\theta_2,\tau_2,\eta_2,k_2) = I(\Gamma,e_1e_2)$$

  By induction hypothesis on $e_1,e_2$:

$$\mathcal{E},s,m \vdash_d e_1 : t_1,m'$$
$$\mathcal{E},s,m' \vdash_d e_2 : t_2,m''$$

  By the soundness proof of the effect-based inference algorithm: $I$ [89]

$$\bar{k}_1\theta_1\Gamma \vdash e_1 : \bar{k}_1\tau_1,\bar{k}_1\eta_1$$
$$\bar{k}_2\theta_2\Gamma \vdash e_1e_2 : \bar{k}_2\tau_2,\bar{k}_2\eta_2$$

  By the definition of the rule (**App**):

$$\mathcal{E},s,m \vdash_d e_1e_2 : \mathsf{M}(\bar{k}_2\tau_2,m'') \cup t_1 \cup t_2 \cup \mathsf{searchTagCall}(e_1,\bar{k}_1\tau_1,t_2,s),m''$$

- **Case of (Let)**

  By the definition of the algorithm:

$$(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$$
$$(t_2,m'') = \mathcal{TG}(\mathcal{E}_x\dagger[x \mapsto t_1],\Gamma,s,m',e_2)$$

  By induction hypothesis on $e_1,e_2$:

$$\mathcal{E},s,m \vdash_d e_1 : t_1,m'$$
$$\mathcal{E}_x\dagger[x \mapsto t_1],s,m' \vdash_d e_2 : t_2,m''$$

  By the definition of the rule (**Let**):

$$\mathcal{E},s,m \vdash_d \textbf{let } x = e_1 \textbf{ in } e_2 : t_2,m''$$

- **Case of (Ref)**

  By the definition of the algorithm:

$$\mathcal{TG}(\mathcal{E},\Gamma,s,m,\textbf{ref}\ e) = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$$

Assume that $(t,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$

By induction hypothesis on $e$:

$$\mathcal{E},s,m \vdash_d e : t,m'$$

By the definition of the rule (**Ref**):

$$\mathcal{E},s,m \vdash_d \textbf{ref}\ (e) : t,m'$$

- **Case of (Deref)**

  By the definition of the algorithm:

$$(t,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e)$$
$$(\theta_1,\tau_1,\eta_1,k_1) = I(\Gamma,e)$$
$$(\theta_2,\tau_2,\eta_2,k_2) = I(\Gamma,!e)$$

  By induction hypothesis on $e$:

$$\mathcal{E},s,m \vdash_d e : t,m'$$

  By the soundness proof of the effect-based inference algorithm $I$ [89]:

$$\bar{k}_1\theta_1\Gamma \vdash e : \bar{k}_1\tau_1,\bar{k}_1\eta_1$$
$$\bar{k}_2\theta_2\Gamma \vdash !e : \bar{k}_2\tau_2,\bar{k}_2\eta_2$$

  By the definition of the rule (**Deref**):

$$\mathcal{E},s,m \vdash_d !e : \mathsf{M}(\bar{k}_2\tau_2,m') \cup t \cup \mathsf{searchTagGet}(e,\bar{k}_1\tau_1,t,s),m''$$

- **Case of (Assign)**

  By the definition of the algorithm:

$$(t_1,m') = \mathcal{TG}(\mathcal{E},\Gamma,s,m,e_1)$$
$$(t_2,m'') = \mathcal{TG}(\mathcal{E},\Gamma,s,m',e_2)$$
$$(\theta,\tau,\eta,k) = I(\Gamma,e_1)$$

  By induction hypothesis on $e_1,e_2$:

$$\mathcal{E},s,m \vdash_d e_1 : t_1,m'$$
$$\mathcal{E},s,m' \vdash_d e_2 : t_2,m''$$

If $\bar{k}\tau = ref_\gamma(\tau')$ then by the soundness proof of the effect-based inference algorithm $I$ [89]:

$$\bar{k}\theta\Gamma \vdash e_1 : ref_\gamma(\tau'), \bar{k}\eta$$

By the definition of the rule (**Assign**):

$$\mathcal{E}, s, m \vdash_d e_1 := e_2 : t, m''_\gamma \dagger [\gamma \mapsto t_2]$$

where

$$t = t_1 \cup t_2 \cup \mathsf{searchTagSet}(e_1, ref_\gamma(\tau'), t_2, s)$$

# 6.3 Type-Based Weaving

In this section, we use the effect-based type inference system to handle the weaving process as it appears in Figure 6.6. For this purpose, we define a new judgment as follows:

$$\Gamma, s \vdash e : \tau, \eta \rightsquigarrow e'$$

This new judgment states that the expression $e$ has type $\tau$ and effect $\eta$ under some typing environment $\Gamma$ and it is translated to $e'$. The translated expression $e'$ is the weaving outcome that results when the applicable advices of the sequence $s$ are weaved into the expression $e$. An advice in $s$ is said to be applicable to $e$ if its pointcut matches $e$.

In the rules (**T-const**) and (**T-var**), the translation makes no changes because there are no applicable advices to weave. In the rules (**T-abs**), (**T-seq**), (**T-letrec**), (**T-let**), (**T-ref**), and (**T-if**) there are also no applicable advices however these rules keep the fact that sub-expressions may have been translated at previous steps. The rules (**T-app**), (**T-deref**), and (**T-assign**) are crucial because we want to pick out join points where we call a function, get a variable, or set a variable. Besides, these join points may match the defined data flow pointcuts. It is essential at these rules to check if any pointcut matches those join points. In case of matching, the applicable advices are injected according to their kinds. We assume that the advices are sorted in the sequence $s$ according to their precedence. The functions $\mathsf{f}_{app}$ in the rule (**T-app**) picks out all the applicable advices that their pointcuts match an application expression. These pointcuts are call pointcuts, data flow pointcuts, or a logical combination between them using boolean operators. Matching a call pointcut depends on the name of a function and its type whereas matching a data flow pointcut in this case depends whether the application expression matches its second pointcut and whether the argument of the application expression is tagged with its tag. The functions $\mathsf{f}_{deref}$ in the rule (**T-deref**) picks out

$$\frac{\mathsf{TypeOf}(c) \succ \tau}{\Gamma, s \vdash c : \tau, \emptyset \rightsquigarrow c} \quad \textbf{(T-const)}$$

$$\frac{x : \sigma \in \Gamma \quad \sigma \succ \tau}{\Gamma, s \vdash x : \tau, \emptyset \rightsquigarrow x} \quad \textbf{(T-var)}$$

$$\frac{\Gamma_x \dagger [x \mapsto \tau_1], s \vdash e : \tau_2, \eta \rightsquigarrow e' \quad \Gamma \vdash \lambda x.e' : \tau', \eta'}{\Gamma, s \vdash \lambda x.e : \tau_1 \xrightarrow{\eta} \tau_2, \emptyset \rightsquigarrow \lambda x.e'} \quad \textbf{(T-abs)}$$

$$\frac{\begin{array}{c}\Gamma, s \vdash e_1 : \tau_1 \xrightarrow{\eta} \tau_2, \eta' \rightsquigarrow e_1' \quad \Gamma, s \vdash e_2 : \tau_1, \eta'' \rightsquigarrow e_2' \\ \mathcal{E}, s, m \vdash_d e_2 : t \quad s' = \mathsf{f}_{app}(e_1, \tau_1 \xrightarrow{\eta} \tau_2, t, s) \\ \langle e_1' e_2', s' \rangle \hookrightarrow \langle e', \varepsilon \rangle \quad \Gamma \vdash e' : \tau, \eta'''\end{array}}{\Gamma, s \vdash e_1 e_2 : \tau_2, ((\eta; \eta'); \eta'') \rightsquigarrow e'} \quad \textbf{(T-app)}$$

$$\frac{\Gamma, s \vdash e_1 : \tau_1, \eta \rightsquigarrow e_1' \quad \Gamma, s \vdash e_2 : \tau_2, \eta' \rightsquigarrow e_2' \quad \Gamma \vdash e_1'; e_2' : \tau', \eta''}{\Gamma, s \vdash e_1; e_2 : \tau_2, (\eta; \eta') \rightsquigarrow e_1'; e_2'} \quad \textbf{(T-seq)}$$

$$\frac{\begin{array}{c}\Gamma_{x,f} \dagger [x \mapsto \tau_1, f \mapsto \tau_1 \xrightarrow{\eta} \tau], s \vdash e_1 : \tau, \eta \rightsquigarrow e_1' \\ \Gamma_f \dagger [f \mapsto Gen(\Gamma, \tau_1 \xrightarrow{\eta} \tau, \eta)], s \vdash e_2 : \tau_2, \eta' \rightsquigarrow e_2' \\ e' = \textbf{let rec } f\ x = e_1' \textbf{ in } e_2' \quad \Gamma \vdash e' : \tau_3, \eta''\end{array}}{\Gamma, s \vdash \textbf{let rec } f\ x = e_1 \textbf{ in } e_2 : \tau_2, (\eta; \eta') \rightsquigarrow e'} \quad \textbf{(T-letrec)}$$

$$\frac{\Gamma, s \vdash e_1 : \tau_1, \eta \rightsquigarrow e_1' \quad \Gamma_x \dagger [x \mapsto Gen(\Gamma, \tau_1, \eta)], s \vdash e_2 : \tau_2, \eta' \rightsquigarrow e_2' \quad \Gamma \vdash \textbf{let } x = e_1' \textbf{ in } e_2' : \tau, \eta''}{\Gamma, s \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2, (\eta; \eta') \rightsquigarrow \textbf{let } x = e_1' \textbf{ in } e_2'} \quad \textbf{(T-let)}$$

$$\frac{\Gamma, s \vdash e : \tau, \eta \rightsquigarrow e'}{\Gamma, s \vdash \textbf{ref } (e) : ref_\rho\ (\tau), (\eta; init(\rho, \tau)) \rightsquigarrow \textbf{ref } (e')} \quad \textbf{(T-ref)}$$

$$\frac{\begin{array}{c}\Gamma, s \vdash e : ref_\rho\ (\tau), \eta \rightsquigarrow e' \quad \mathcal{E}, s, m \vdash_d e : t \quad s' = \mathsf{f}_{deref}(e, ref_\rho\ (\tau), t, s) \\ \langle !e', s' \rangle \hookrightarrow \langle e_1, \varepsilon \rangle \quad \Gamma \vdash e_1 : \tau_1, \eta'\end{array}}{\Gamma, s \vdash\ !e : \tau, (\eta; read(\rho, \tau)) \rightsquigarrow e_1} \quad \textbf{(T-deref)}$$

$$\frac{\begin{array}{c}\Gamma, s \vdash e_1 : ref_\rho\ (\tau_1), \eta \rightsquigarrow e_1' \quad \Gamma, s \vdash e_2 : \tau_1, \eta' \rightsquigarrow e_2' \\ \mathcal{E}, s, m \vdash_d e_2 : t \quad s' = \mathsf{f}_{assign}(e_1, ref_\rho\ (\tau_1), t, s) \\ \langle e_1' := e_2', s' \rangle \hookrightarrow \langle e', \varepsilon \rangle \\ \Gamma \vdash e' : \tau, \eta''\end{array}}{\Gamma, s \vdash e_1 := e_2 : unit, ((\eta; \eta'); write(\rho, \tau_1)) \rightsquigarrow e'} \quad \textbf{(T-assign)}$$

Figure 6.6: Type-Based Weaving Rules

all the applicable advices that their pointcuts match a dereferncing expression. These pointcuts are get pointcuts, data flow pointcuts, or a logical combination between them using boolean operators. Matching a get pointcut depends on the name of a variable and its type whereas matching a data flow pointcut in this case depends on whether the dereferncing expression matches its second pointcut and whether the argument of the dereferncing expression is tagged with its tag. The functions $\mathbf{f}_{assign}$ in the rule (**T-assign**) picks out all the applicable advices that their pointcuts match an assignment expression. These pointcuts are set pointcuts, data flow pointcuts, or a logical combination between them using boolean operators. Matching a set pointcut depends on the name of a variable and its type whereas matching a data flow pointcut in this case depends on whether the assignment expression matches its second pointcut and whether the right-hand side of the assignment expression is tagged with its tag.

In Figure 6.6, the weaving configuration is represented by $\langle Exp, AdvSeq \rangle$. Hence, the rule $\langle e,s \rangle \hookrightarrow \langle e',\varepsilon \rangle$ means that $e'$ is the result of weaving all the advices in $s$ into $e$. Notice that $\hookrightarrow$ is transitive. Hereafter, we give the weaving rules:

- **Rule 1**: *Before advice weaving*

$$\frac{s = as' \quad a.kind = \texttt{before}}{\langle e,s \rangle \hookrightarrow \langle a.exp; e, s' \rangle}$$

- **Rule 2:** *After advice weaving*

$$\frac{s = as' \quad a.kind = \texttt{after}}{\langle e,s \rangle \hookrightarrow \langle \textbf{let } tmp = e \textbf{ in } a.exp; tmp, s' \rangle}$$

- **Rule 3:** *Around advice weaving without proceed*

$$\frac{\begin{array}{c} s = as' \quad a.kind = \texttt{around} \quad \Gamma \vdash e : \tau, \eta \\ \Gamma \vdash a.exp : \tau', \eta' \quad \theta\tau = \theta\tau' \quad \theta\eta \subseteq \theta\eta' \\ \mathsf{containProceed}(a.exp) = false \end{array}}{\langle e,s \rangle \hookrightarrow \langle a.exp, s' \rangle}$$

- **Rule 4:** *Around advice weaving with proceed*

$$\frac{\begin{array}{c} s = as' \quad a.kind = \texttt{around} \\ \langle e, s' \rangle \hookrightarrow \langle e', \varepsilon \rangle \quad \Gamma \vdash e : \tau, \eta \\ \Gamma \vdash a.exp\ [proceed \mapsto e'] : \tau', \eta' \\ \theta\tau = \theta\tau' \quad \theta\eta \subseteq \theta\eta' \\ \mathsf{containProceed}(a.exp) = true \end{array}}{\langle e, s \rangle \hookrightarrow \langle a.exp\ [proceed \mapsto e'], \varepsilon \rangle}$$

The weaving process is in the spirit of AspectJ. The sequence construct ";" of the extended λ-calculus is introduced to perform the injection. The before-advice is inserted before the expressions that match its pointcut. Similarly, the after-advice is inserted after the expressions that match its pointcut. Actually, the value of the matched expression should be returned after executing the matched expression inside the advice body. The around-advice bypasses the computation of a join point. The around-advice with *proceed* allows to run the advice with next precedence, or the computation under the join point if there is no further advice. Besides, the type of the around-advice must be the same or an instance of the type of the expression that matches its pointcut. In the following, we state and prove a result that establishes the preservation of the weaving process.

**Theorem: Preservation**

If $\Gamma \vdash e : \tau, \eta$ and $\langle e, s \rangle \hookrightarrow \langle e', \varepsilon \rangle$ then $\Gamma \vdash e' : \tau', \eta'$ where there exists a substitution $\theta$ such that $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$.

**Proof**

The proof is done by induction over the length of $s$. Assume that $|s| = k$.

1. Induction basis ($k = 0$).

   $\Gamma \vdash e : \tau, \eta$ and $\langle e, \varepsilon \rangle \hookrightarrow \langle e', \varepsilon \rangle$. By the weaving rules, $e = e'$. Consequently, the implication is satisfied by taking $\theta = id$.

2. Induction hypothesis ($0 \leq k \leq n$).

   $$\left.\begin{array}{l} \Gamma \vdash e : \tau, \eta \\ \langle e, s \rangle \hookrightarrow \langle e', \varepsilon \rangle \end{array}\right\} \rightarrow \Gamma \vdash e' : \tau', \eta'$$

   where there exists a substitution $\theta$ such that $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$.

3. Induction step ($k = n + 1$).

   let's assume $s' = as$ where $|as| = n + 1$, we have to prove that

$$\left.\begin{array}{l}\Gamma \vdash e : \tau, \eta \\ \langle e, as \rangle \hookrightarrow \langle e', \varepsilon \rangle\end{array}\right\} \rightarrow \Gamma \vdash e' : \tau', \eta'$$

where there exists a substitution $\theta$ such that $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$.

- **Case ($a$ is a before-advice)**

  By the first rule of the weaving rules

  $$\langle e, as \rangle \hookrightarrow \langle a.exp; e, s \rangle$$

  Then

  $$\langle a.exp; e, s \rangle \hookrightarrow \langle e', \varepsilon \rangle$$

  By assumption

  $$\Gamma \vdash e : \tau, \eta$$

  By the typing rule of the sequence construct

  $$\Gamma \vdash a.exp; e : \tau, \eta'' \text{ where } \eta \subseteq \eta''$$

  By hypothesis

  $$\left.\begin{array}{l}\Gamma \vdash a.exp; e : \tau, \eta'' \\ \langle a.exp; e, s \rangle \hookrightarrow \langle e', \varepsilon \rangle\end{array}\right\} \rightarrow \Gamma \vdash e' : \tau', \eta'$$

  where there exists a substitution $\theta$ such that $\theta\tau = \theta\tau'$ and $\theta\eta'' \subseteq \theta\eta'$.

  Since $\eta \subseteq \eta''$ and $\theta\eta'' \subseteq \theta\eta'$ then $\theta\eta \subseteq \theta\eta'$ and finally $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$.

- **Case ($a$ is an after-advice)**

  By the second rule of the weaving rules

  $$\langle e, as \rangle \hookrightarrow \langle \textbf{let } tmp = e \textbf{ in } a.exp; tmp, s \rangle$$

  Then

  $$\langle \textbf{let } tmp = e \textbf{ in } a.exp; tmp, s \rangle \hookrightarrow \langle e', \varepsilon \rangle$$

  By assumption

  $$\Gamma \vdash e : \tau, \eta$$

  By the typing rule of the sequence construct

$$\Gamma \vdash \textbf{let } tmp = e \textbf{ in } a.exp; tmp : \tau, \eta'' \text{ where } \eta \subseteq \eta'' \text{ and } tmp \text{ is not free in } a.exp$$

By hypothesis

$$\left.\begin{array}{l} \Gamma \vdash \textbf{let } tmp = e \textbf{ in } a.exp; tmp : \tau, \eta'' \\ \langle \textbf{let } tmp = e \textbf{ in } a.exp; tmp, s\rangle \hookrightarrow \langle e', \varepsilon\rangle \end{array}\right\} \rightarrow \Gamma \vdash e' : \tau', \eta'$$

where there exists a substitution $\theta$ such that $\theta\tau = \theta\tau'$ and $\theta\eta'' \subseteq \theta\eta'$.

Since $\eta \subseteq \eta''$ and $\theta\eta'' \subseteq \theta\eta'$ then $\theta\eta \subseteq \theta\eta'$ and finally $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$.

- **Case (*a* is an around-advice without *proceed*)**

  By the third rule of the weaving rules

  $$\langle e, as\rangle \hookrightarrow \langle a.exp, s\rangle$$

  Then

  $$\langle a.exp, s\rangle \hookrightarrow \langle e', \varepsilon\rangle$$

  By assumption

  $$\Gamma \vdash e : \tau, \eta$$

  By the third rule of the weaving rules

  $$\Gamma \vdash a.exp : \tau'', \eta'' \text{ where } \theta_1\tau = \theta_1\tau'' \text{ and } \theta_1\eta \subseteq \theta_1\eta''.$$

  By hypothesis

  $$\left.\begin{array}{l} \Gamma \vdash a.exp : \tau'', \eta'' \\ \langle a.exp, s\rangle \hookrightarrow \langle e', \varepsilon\rangle \end{array}\right\} \rightarrow \Gamma \vdash e' : \tau', \eta'$$

  where there exists a substitution $\theta_2$ such that $\theta_2\tau'' = \theta_2\tau'$ and $\theta_2\eta'' \subseteq \theta_2\eta'$.

  Since $\theta_1\tau = \theta_1\tau''$ and $\theta_2\tau'' = \theta_1\tau'$, then there is a substitution $\theta = \theta_2\theta_1$ such that $\theta\tau' = \theta\tau$.

  Similarly, since $\theta_1\eta \subseteq \theta_1\eta''$ and $\theta_2\eta'' \subseteq \theta_2\eta'$ then $\theta\eta \subseteq \theta\eta'$.

- **Case (*a* is around-advice with *proceed*)**

  By the fourth rule of the weaving rules

  $$\langle e, as\rangle \hookrightarrow \langle a.exp\,[proceed \mapsto e''], \varepsilon\rangle$$

Where

$$\langle e, s' \rangle \hookrightarrow \langle e'', \varepsilon \rangle$$

Then

$$\langle a.exp\ [proceed \mapsto e''], \varepsilon \rangle \hookrightarrow \langle e', \varepsilon \rangle$$

By assumption

$$\Gamma \vdash e : \tau, \eta$$

By the fourth rule of the weaving rules

$$\Gamma \vdash a.exp\ [proceed \mapsto e''] : \tau'', \eta'' \text{ where } \theta\tau = \theta\tau'' \text{ and } \theta\eta \subseteq \theta\eta''$$

By the weaving rules

$$e' = a.exp\ [proceed \mapsto e'']$$

Consequently, if we have $\Gamma \vdash a.exp\ [proceed \mapsto e''] : \tau'', \eta''$ and $\Gamma \vdash e' : \tau', \eta'$ then $\tau'$ is equal to $\tau''$ and $\eta'$ is equal to $\eta''$. Consequently, $\theta\tau = \theta\tau'$ and $\theta\eta \subseteq \theta\eta'$ .

**Example**

We present in Figure 6.7 the derivation according to the type-based weaving rules for the following expression, advice, and pointcut:

**Expression:**

$e = (\textbf{let rec}\, fx = x \,\textbf{in}\, f2)$

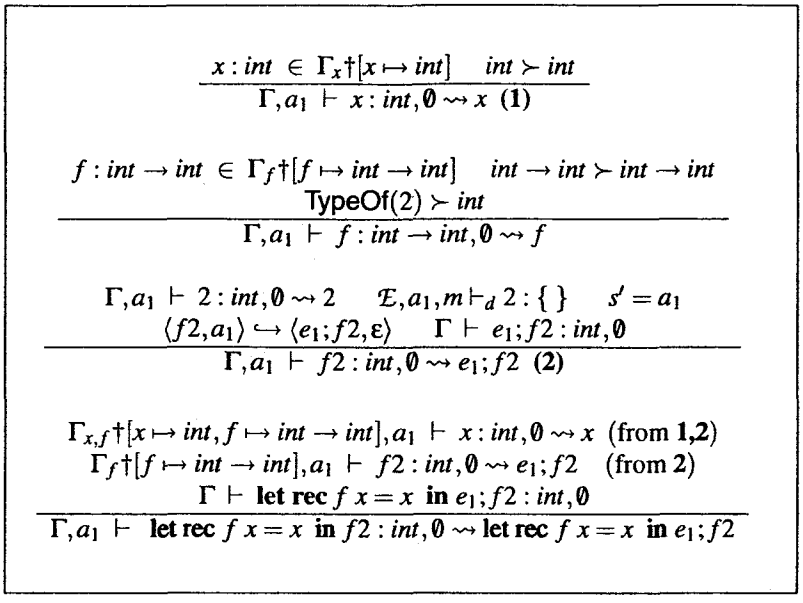
**Advices:**

$a_1$::= ⟨*akind*: `before`, *pcd*:$p_1$, *exp*:$e_1$ ⟩

**Pointcuts:**

$p_1$::= ⟨*pkind*: `call`, *var*:*f*, *typeScheme*: $\forall\alpha.\alpha \rightarrow \alpha$ ⟩

**Derivation:** The rules (**T-const**), (**T-var**), (**T-app**), and (**T-letrec**) are used in the derivation. The matching and the weaving of the advice $a_1$ happen during typing the expression $f2$.



$$\frac{x : int \in \Gamma_x\dagger[x \mapsto int] \quad int \succ int}{\Gamma, a_1 \vdash x : int, \emptyset \rightsquigarrow x \ \textbf{(1)}}$$

$$\frac{f : int \to int \in \Gamma_f\dagger[f \mapsto int \to int] \quad int \to int \succ int \to int \quad \mathsf{TypeOf}(2) \succ int}{\Gamma, a_1 \vdash f : int \to int, \emptyset \rightsquigarrow f}$$

$$\frac{\Gamma, a_1 \vdash 2 : int, \emptyset \rightsquigarrow 2 \quad \mathcal{E}, a_1, m \vdash_d 2 : \{\,\} \quad s' = a_1 \quad \langle f2, a_1 \rangle \hookrightarrow \langle e_1; f2, \varepsilon \rangle \quad \Gamma \vdash e_1; f2 : int, \emptyset}{\Gamma, a_1 \vdash f2 : int, \emptyset \rightsquigarrow e_1; f2 \ \textbf{(2)}}$$

$$\frac{\Gamma_{x,f}\dagger[x \mapsto int, f \mapsto int \to int], a_1 \vdash x : int, \emptyset \rightsquigarrow x \ (\text{from } \textbf{1,2}) \quad \Gamma_f\dagger[f \mapsto int \to int], a_1 \vdash f2 : int, \emptyset \rightsquigarrow e_1; f2 \ (\text{from } \textbf{2}) \quad \Gamma \vdash \textbf{let rec } f\ x = x \textbf{ in } e_1; f2 : int, \emptyset}{\Gamma, a_1 \vdash \textbf{let rec } f\ x = x \textbf{ in } f2 : int, \emptyset \rightsquigarrow \textbf{let rec } f\ x = x \textbf{ in } e_1; f2}$$

Figure 6.7: Example of Derivations

## 6.4 Inference Algorithm

The present section is dedicated to the algorithm $I\mathcal{W}$ of the type-based weaving rules presented in Figure 6.6. The algorithm $I\mathcal{W}$ presented next proceeds by case analysis on the structure of expressions. It takes as input a 5-tuple made of a static environment $\Gamma$, a tagging environment $\mathcal{E}$, a sequence of defined advices $s$, a store that maps regions to tag sets $m$, and an expression $e$. The algorithm either fails or terminates successfully producing a 5-tuple whose components are: a substitution $\theta$, a type $\tau$, an effect $\eta$, a set of constraints $k$, and the weaved expression $e'$. It applies the algorithm $I$ of Talpin and Jouvelot presented in Section 2.4.4. The algorithm is written for the basic expressions but the rest of the cases are straightforward. The algorithm is defined as follows:

$I\mathcal{W}(\Gamma, \mathcal{E}, s, m, c) =$
**let** $\forall \upsilon_1, ..., \upsilon_n.(\tau, k) = CTypeOf(c)$
  $\upsilon'_1, ..., \upsilon'_n$ **new**, $\theta = [\upsilon_1 \mapsto \upsilon'_1, ..., \upsilon_n \mapsto \upsilon'_n]$
**in** $(id, \theta\tau, \emptyset, \theta k, c)$

$I\mathcal{W}(\Gamma, \mathcal{E}, s, m, x) =$
**if** $x \notin \mathrm{Dom}(\Gamma)$ **then** *fail*
**else**
 **let** $\forall \upsilon_1, ..., \upsilon_n.(\tau, k) = \Gamma(x)$

$\upsilon'_1,\ldots,\upsilon'_n$ **new**, $\theta = [\upsilon_1 \mapsto \upsilon'_1,\ldots,\upsilon_n \mapsto \upsilon'_n]$

**in** $(id, \theta\tau, \emptyset, \theta k, x)$

$I\mathcal{W}(\Gamma,\mathcal{E},s,m,\lambda x.e) =$

**let** $\alpha,\varsigma$ **new**

$(\theta_1,\tau_1,\eta_1,k_1,e') = I\mathcal{W}(\Gamma_x\dagger[x\mapsto\alpha],\mathcal{E},s,m,e)$

$(\theta_2,\tau_2,\eta_2,k_2) = I(\Gamma,\lambda x.e')$

**in** $(\theta_1,\theta_1\alpha \xrightarrow{\varsigma} \tau_1,\emptyset,k_1\cup\{\eta_1\subseteq\varsigma\},\lambda x.e')$

$I\mathcal{W}(\Gamma,\mathcal{E},s,m,e_1e_2) =$

**let** $(\theta_1,\tau_1,\eta_1,k_1,e'_1) = I\mathcal{W}(\Gamma,\mathcal{E},s,m,e_1)$

$(\theta_2,\tau_2,\eta_2,k_2,e'_2) = I\mathcal{W}(\theta_1\Gamma,\mathcal{E},s,m,e_2)$

$\alpha,\varsigma$ **new**, $\theta_3 = \mathcal{U}(\theta_2\tau_1,\tau_2 \xrightarrow{\varsigma} \alpha)$

$s' = \mathsf{f}_{app}(e_1,\tau_1,\mathcal{TG}(\mathcal{E},\Gamma,s,m,e_2),s)$

$e' = \mathcal{WV}(\Gamma,e'_1e'_2,s')$

$(\theta_4,\tau_4,\eta_4,k_4) = I(\Gamma,e')$

**in** $(\theta_3\theta_2\theta_1,\theta_3\alpha,\theta_3(\theta_2\eta_1;\eta_2;\varsigma),\theta_3(\theta_2k_1\cup k_2),e')$

$I\mathcal{W}(\Gamma,\mathcal{E},s,m,\textbf{let } x = e_1 \textbf{ in } e_2) =$

**let** $(\theta_1,\tau_1,\eta_1,k_1,e'_1) = I\mathcal{W}(\Gamma,\mathcal{E},s,m,e_1)$

$(\theta_2,\tau_2,\eta_2,k_2,e'_2) =$

$I\mathcal{W}(\theta_1\Gamma_x\dagger[x\mapsto Gen_{k_1}(\theta_1\Gamma,\tau_1,\eta_1)],\mathcal{E},s,m,e_2)$

$(\theta_3,\tau_3,\eta_3,k_3) = I(\Gamma,\textbf{let } x = e'_1 \textbf{ in } e'_2)$

**in** $(\theta_2\theta_1,\tau_2,\theta_2\eta_1;\eta_2,\theta_2k_1\cup k_2,\textbf{let } x = e'_1 \textbf{ in } e'_2)$

$I\mathcal{W}(\Gamma,\mathcal{E},s,m,\textbf{ref } e) =$

**let** $\gamma$ **new**

$(\theta,\tau,\eta,k,e') = I\mathcal{W}(\Gamma,\mathcal{E},s,m,e)$

**in** $(\theta, ref_\gamma(\tau),\eta;init(\gamma,\tau),k,\textbf{ref } e')$

$I\mathcal{W}(\Gamma,\mathcal{E},s,m,!e) =$

**let** $(\theta_1,\tau_1,\eta_1,k_1,e') = I\mathcal{W}(\Gamma,\mathcal{E},s,m,e)$

$\alpha,\varsigma$ **new**, $\theta_2 = \mathcal{U}(ref_\gamma(\alpha),\tau_1)$

$s' = \mathsf{f}_{deref}(e,\tau_1,\mathcal{TG}(\mathcal{E},\Gamma,s,m,e),s)$

$e_1 = \mathcal{WV}(\Gamma, !e', s')$

$(\theta_3, \tau_3, \eta_3, k_3) = I(\Gamma, e_1)$

**in** $(\theta_2\theta_1, \theta_2\alpha, \eta_1; read(\theta_2\gamma, \theta_2\alpha), \theta_2 k_1, e_1)$

$I\mathcal{W}(\Gamma, \mathcal{E}, s, m, e_1 := e_2) =$

**let** $(\theta_1, \tau_1, \eta_1, k_1, e_1') = I\mathcal{W}(\Gamma, \mathcal{E}, s, m, e_1)$

$(\theta_2, \tau_2, \eta_2, k_2, e_2') = I\mathcal{W}(\theta_1\Gamma, \mathcal{E}, s, m, e_2)$

$\gamma$ **new**

$\theta_3 = \mathcal{U}(ref_\gamma(\tau_2), \theta_2\tau_1)$

$s' = \mathrm{f}_{assign}(e_1, \tau_1, \mathcal{TG}(\mathcal{E}, \Gamma, s, m, e_2), s)$

$e' = \mathcal{WV}(\Gamma, e_1' := e_2', s')$

$(\theta_4, \tau_4, \eta_4, k_4) = I(\Gamma, e')$

**in** $(\theta_3\theta_2\theta_1, unit, \theta_3(\theta_2\eta_1; \eta_2; write(\gamma, \tau_2)),$

$\theta_3(\theta_2 k_1 \cup k_2), e')$

**Theorem: Soundness**

Given a typing environment $\Gamma$, a tagging environment $\mathcal{E}$, a sequence of defined advices $s$, a store that maps regions to tag sets $m$, and an expression $e$. If $I\mathcal{W}(\Gamma, \mathcal{E}, s, m, e) = (\theta, \tau, \eta, k, e')$, then $\bar{k}\theta\Gamma, s \vdash e : \bar{k}\tau, \bar{k}\eta \rightsquigarrow e'$.

**Proof**

The proof is standard and can be found in the contribution of Talpin and Jouvelot [89].

You can notice that we do not extend the effect typing system. We type an expression according to the original algorithm $I$, search for all the applicable advices for this expression and weave them with it, and then type the weaved expression that result from the weaving process according to the original algorithm $I$. Accordingly, the soundness proof of the previous contributions [89] are applicable here. What we need to add is the soundness proof for the tagging algorithm $\mathcal{TG}$ and that what we did in Section 6.2.

## 6.5 Conclusion

In this chapter, we have presented a security aspect oriented calculus: λ_SAOP based on the well-known λ_calculus together with the semantic foundations. This will be a very useful and precious contribution to the users/researchers in the field of Aspect Oriented Programming and those who are interested in solving security problems. The calculus contains useful pointcuts from a security perspective such as `call`, `get`,

`set`, and `dflow` pointcuts. The advice weaving in λ_SAOP is in the spirit of AspectJ, a prominent Aspect Oriented Programming language. The weaving is done during the typing process and uses tags to match data flow pointcuts.

This contribution is the first step towards a complete security aspect core based on the extended λ-calculus together with the semantic foundations.

# Chapter 7

# Design and Implementation of AspectJ Extensions

In this chapter, the extension of the Eclipse AspectJ compiler `ajc` version 1.5 is described. This extension consists of designing and implementing new pointcuts that are relevant from a security point of view. The considered pointcuts are related to local variable accesses, namely `getLocal` and `setLocal`, and to data flow information analysis, namely `dflow`. The appropriateness of these pointcuts for software security hardening has been discussed in Chapter 3. Section 7.1 presents the implemented pointcuts. Section 7.2 gives an overview of Eclipse AspectJ compiler `ajc` architecture. Section 7.3 describes `ajc` software package details. Finally Section 7.4 presents the implementation.

## 7.1 Design of the Proposed Pointcuts

Basic pointcuts in AspectJ fall into three classes:

- **Kinded pointcuts:** This class of pointcuts corresponds to join points that match directly a granular bytecode instruction or a set of bytecode instructions. For example, a call pointcut matches an invoke bytecode instruction whereas an execution pointcut matches all the bytecode instructions bound to a method or a constructor execution.

- **Scope matching pointcuts:** This class of pointcuts designators targets a set of join points within a certain scope in the program. We distinguish two kinds of scopes, a static scope and a dynamic scope. A static scope is a syntactic location in a program, as a class, or a package. A dynamic scope is a program runtime location, as a control flow of a method call or execution. The aim of such pointcuts is to limit join points location lookup inside a program.
- **Context matching pointcuts:** This class of pointcuts designators focuses on providing contextual information such runtime object values. Such pointcuts are generally used in conjunction with kinded pointcuts.

In this section we give an overview of the new pointcuts: `getLocal`, `setLocal`, and `dflow`. We present the syntax and the informal semantics of each pointcut.

### 7.1.1 Pointcuts: `getlocal` and `setlocal`

AspectJ does not allow tracking the variables of local variables inside methods. We have shown in Chapter 3 that these pointcuts are relevant for security because they allow to follow the information flow and to intervene in case of sensitive information disclosure. Furthermore, these pointcuts are extremely important for a complete design and implementation of the data flow pointcut. In fact, the data flow pointcut allows to perform data flow analysis, which would be uncomplete if we do not consider the pointcuts `getLocal` and `setLocal`.

The syntax that we choose for each of the pointcuts is as follows:

- `getlocal` (*MethodPattern*, *LocalVariablePatternList*)
- `setlocal` (*MethodPattern*, *LocalVariablePatternList*)

The pattern *MethodPattern* already exists in AspectJ and allows to describe the targeted methods in the pointcut. In our case, the *MethodPattern* will describe the methods containing the local variables that we want to set or to read. The new pattern *LocalVariablePatternList* is introduced to identify specific local variables in the methods specified by *MethodPattern*. It contains the list of local variables that we want to read/set where each local variable is identified by its name preceded by its type. We can also use the wildcard "*" of AspectJ, which denotes arbitrary strings. Hence, in a local variable pattern list, the wildcard "*" can appear at most twice either in the type part or/and in the name part of the local variable syntax. We present hereafter some examples of local variables pointcuts:

- The pointcut `getlocal (* *.f(..),int *)` will match any join point in any method `f` where an int local variable is read.

- The pointcut `setlocal (* *.f(..),* *)` will match any join point in any method f where a local variable is set.
- The pointcut `setlocal (* *.f(..),int a, float b)` will match the join points in any method f where an int local variable a or a float local variable b is set.

We classify the pointcuts `getLocal` and `setLocal` as kinded pointcuts. The join point shadows for the local variable pointcuts are specific bytecode instructions. The `getlocal` pointcut matches the following bytecode instructions: { `aload`, `aload_<n>`, `dload`, `dload_<n>`, `fload`, `fload_<n>`, `iload`, `iload_<n>`, `lload`, `lload_<n>` } whereas the `setlocal` pointcut matches the bytecode instructions: { `astore`, `astore_<n>`, `dstore`, `dstore_<n>`, `fstore`, `fstore_<n>`, `istore`, `istore_<n>`, `lstore`, `lstore_<n>` }.

### 7.1.2 Pointcut `dflow`

We classify the `dflow` pointcut as a scope matching pointcut like the pointcut `cflow`. Because of `cflow` implementation is done in AspectJ, we choose for the data flow pointcut a syntax close to the syntax of the control flow pointcut. The data flow pointcut has then the following syntax:

`dflow` (*Pointcut*)

The rule that relates together a join point with a data flow pointcut is the following:
*A join point j matches a data flow pointcut* `dflow` *(p) if and only if j is in the data flow of another join point j', which matches p.*

Section 7.4 will describe how the data flow dependencies are handled through the data flow pointcut implementation.

## 7.2 AspectJ Compiler Architecture

The `ajc` compiler is the original compiler of AspectJ and the reference implementation of the language. The main modules of the ajc compiler are illustrated in Figure 7.1.

The front-end compiler is an extended version of Eclipse's JDT compiler. It takes as input AspectJ source codes and returns standard Java class files enriched with extra attributes that handle non-pure Java information as advices and pointcuts. It also generates special names for advice bodies and other methods implementing special AspectJ constructs.

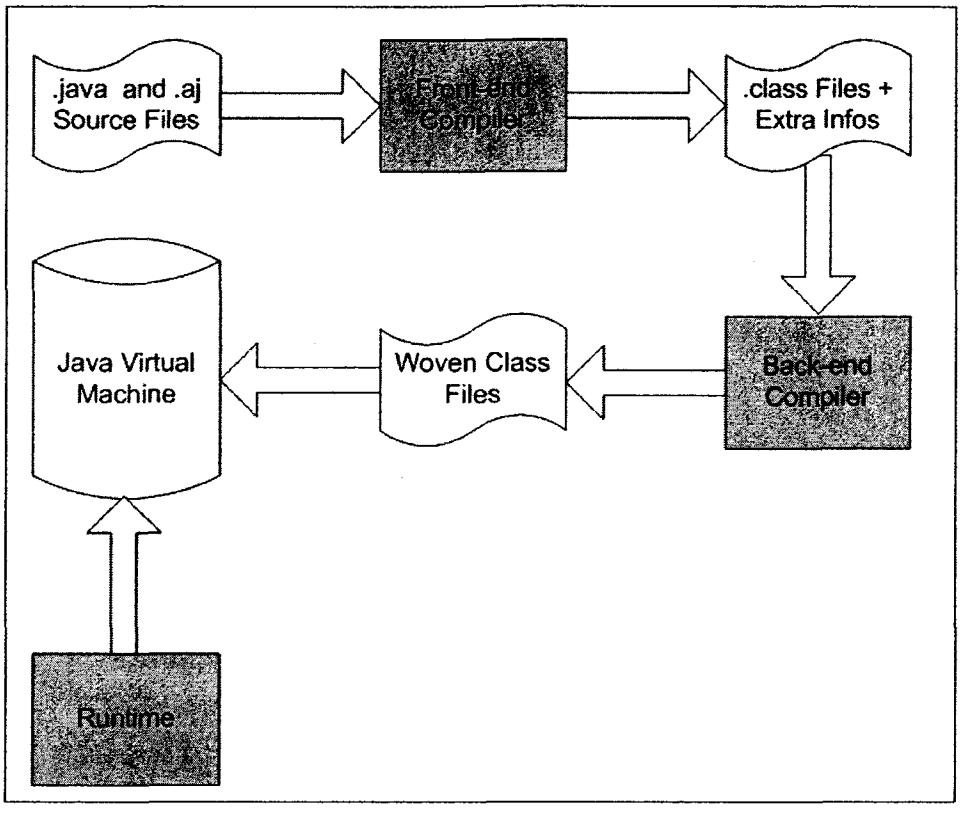



Figure 7.1: AspectJ Compiler Architecture

The back-end-compiler provides the bytecode weaving functionality and weaves compiled aspects with compiled applications producing woven class files.

The runtime is a module that contains classes used by generated code during runtime. These classes need to be redistributed with a system built using AspectJ. It contains, for example, classes that are necessary for the implementation of control flow pointcuts.

We can see, through the example of Figure 7.2, how the weaving modifies the JVML (Java Virtual Machine) code of the function `f`.

The aspect `A` has one before advice that prints "aspect" when its pointcut is matched. The pointcut is matched when the join point is either a method execution or a set field join point. In the method `f`, we have two join points that are matched: the execution of `f` and the update of the field `a`, represented by the JVML instruction: `putfield #2`. Hence, AspectJ weaver injects the advice at the beginning of `f` and before the instruction `putfield #2`. The instructions `invokestatic #26` and `invokevirtual #29` are injected. The instruction `invokestatic #26` calls a static method named `aspectOf`, which is automatically generated by the front-end compiler once the aspect is compiled. It allows obtaining an

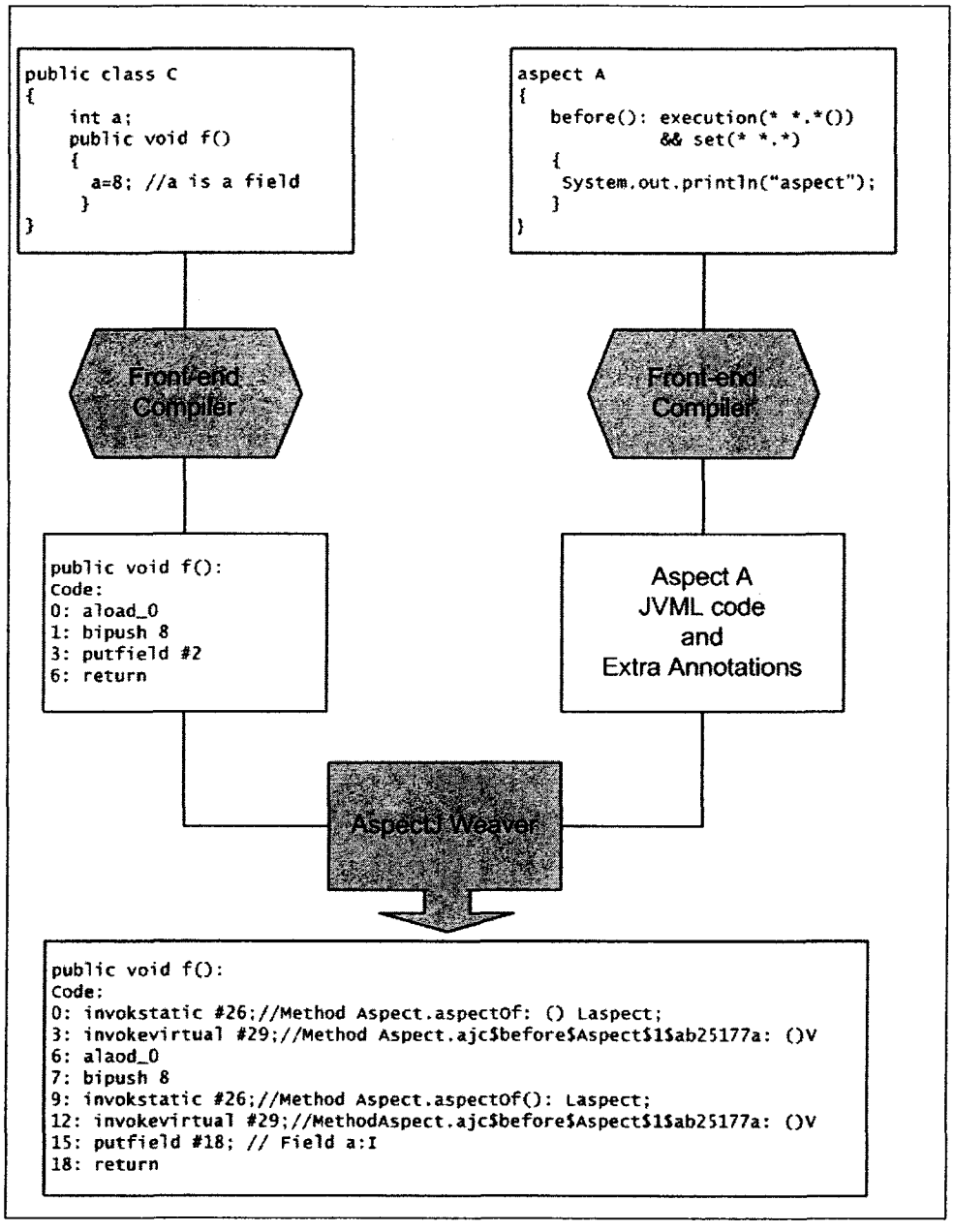



Figure 7.2: Weaving Process

instance of the aspect, which will serve as a receiver for the advice call. In the case of this example, the instance serves as a receiver for the instruction `invokevirtual #29` that corresponds to the advice method call.

### 7.2.1 Front-End Compiler

The principal task of the front-end compiler is to perform a shallow parse on all source files and to build abstract syntax trees for the different declarations and designators provided by AspectJ syntax. An advice declaration is compiled into a standard Java method and the parameters of the new method are the parameters of the advices. Notice that the method is annotated with an additional attribute that indicates that this corresponds to an advice declaration. Furthermore, this attribute allows also to store the pointcut referred to the advice, and to store additional information relevant to the matching and weaving processes. For example, in case of around advice the front-end compiler transmits through this attribute to the back-end compiler the information that the advice body contains or not a call to `proceed`. This attribute is encoded as a standard Java bytecode attribute in order to be compatible with all the JVMs.

### 7.2.2 Back-End Compiler

The main phases of the back-end compiler, as illustrated in Figure 7.3, are: Parsing Pointcut, making shadows, creating shadow mungers, matching, and weaving. We detail in the following each of these phases.

**Pointcut Parsing**

The pointcut parsing phase captures pointcut designator patterns and creates objects that correspond to these pointcuts. A pattern is a syntactic representation that holds information about types, fields, methods, pointcuts, advices and aspects. Figure 7.4 provides a parsing example.

The pattern parser captures the designator `call (* Test.test (..))` and extracts information that is necessary to create a kinded pointcut object that points to the declared pointcut.

The kind is a method call and the signature is considered as a fingerprint that designates a method or a set of methods. In the above example, the signature can match any method that has any return type, a by default public modifier, `Test` class as a declared type, `test` as a method name, any parameters type and can throw any exception.

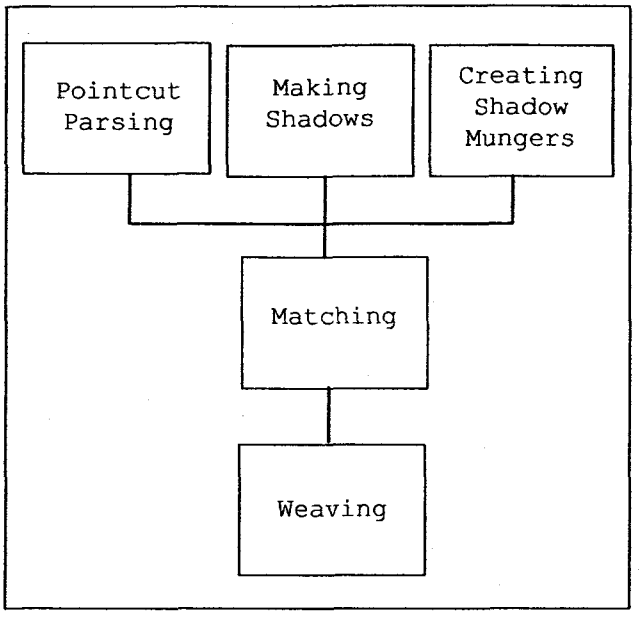



Figure 7.3: Back-End Compiler Phases

**Making Shadows**

This phase defines all the possible static shadows in the program and injects markers in the code in order to wrap these static shadows. In AspectJ implementation, a joinpoint is represented by a bytecode shadow enriched with a kind to distinguish it from other shadows and a member signature such as a method or a field. In other terms, a shadow is a wrapper that surrounds a bytecode instruction or a set of bytecode instructions that symbolizes a dynamic joinpoint. In Figure 7.5, we illustrate how the static shadow of a method-call join point is represented:

The `invokevirtual` bytecode instruction is wrapped by the "method-call" shadow enriched with the method signature `void Test.test ()`.

**Creating Shadow Mungers**

This phase allows representing advice entities by shadow munger objects. Each shadow munger transforms the join point shadows that match its pointcut. During the weaving phase, each join point in the program being processed is compared against each shadow munger pointcut. Because the join points in AspectJ are defined as dynamic points in the call graph of the program, the matching process might need runtime information and might not be completely statically resolved. In the case where the shodow munger pointcut depends on the dynamic state at the join point, this is resolved by adding a dynamic test, called

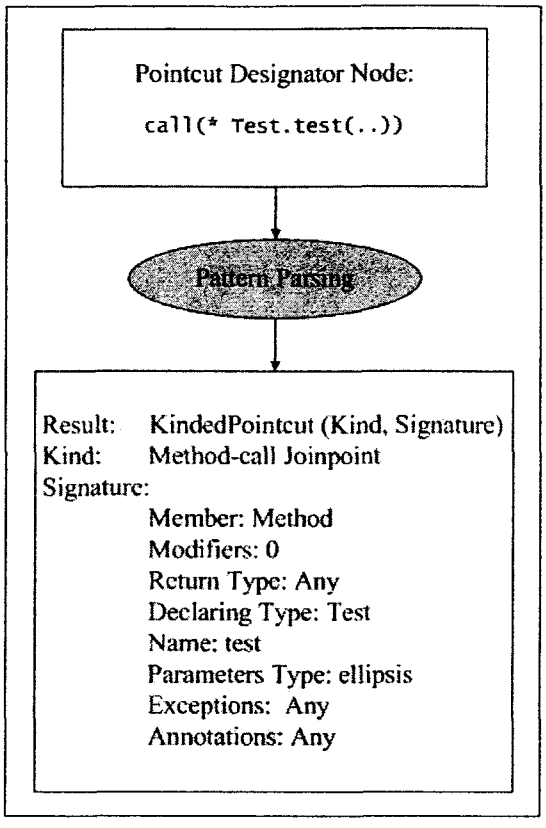



Figure 7.4: Pointcut Parsing Example

```
Method-call(void Test.test())
|    INVOKEVIRTUAL Test.test() V
Method-call(void Test.test())
```

Figure 7.5: Method Call Shadow Representation

residue of the match that describes the dynamic part of the matching.

**Matching**

This phase tries to match each join point shadow against the shadow mungers. There are three possible results for each shadow: never matches, always matches, and sometimes matches. In the first case, nothing will be added to the shadow. In the second case, the advice will be woven into the join point shadow. In the third case the advice is woven into the shadow with the dynamic tests to determine if the joinpoint in the running program matches the advice.

**Weaving**

This phase allows to inject advices inside matched join point shadows and the dynamic tests, called residues, if any. The residues falls into three kinds:

- **Residue** `instanceof`: This kind of residue is generated when the pointcuts `args`, `this` or `target` pointcuts are used and the matching result is "sometimes matches". This residue allows to check whether an object is an instance of the type defined in the pointcut.
- **Residue** `If`: This residue holds when the `if` pointcut designator is used to provide a boolean result regarding evaluation of an expression at matched joinpoints.
- **Control flow Residue**: This residue is computed if the `cflow` pointcut designator is used inside an advice declaration. This test is based on a thread local stack or a counter to keep track of the control flow. The counter is used, for optimization, if the `cflow` entry pointcut does not bind any values.

## 7.3 AspectJ Open Source Software Package Details

This section presents some details of the most important `ajc` packages. Figure 7.6 represents the packages corresponding to the different phases of the compiler architecture. We can mention three important modules: 1) The module *org.aspectj.ajdt.core*, which is the front-end compiler and extends the eclipse Java compiler, 2) The *runtime* module that provides classes that are used by the generated code at runtime, and 3) The *weaver* module, which offers the weaving functionality.

We will describe the packages contained in the modules *runtime* and *weaver*. We will not present the module *org.aspectj.ajdt.core* since it is just an extension of the JDT Core *org.eclipse-.jdt.core*, which is the plug-in that defines the core Java elements and API.

### 7.3.1 Runtime packages

We describe in the following the different packages in the module *runtime*:

- Package *org.aspectj.lang*: Provides a set of classes for interacting with join points. More precisely, this package focuses on making information about matched join points that are available within the advice body through the special variables such as `thisJoinPoint`. It is similar to the familiar *java.lang* package, which contains the most important constructs for the Java language.

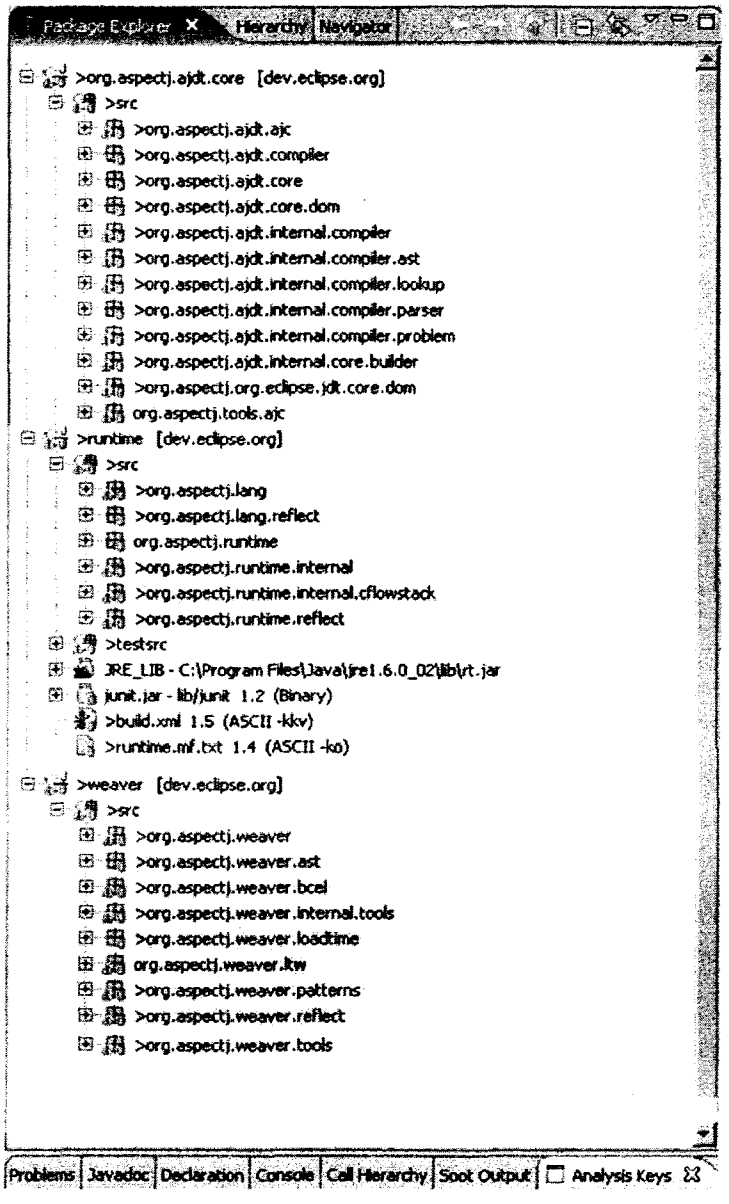



Figure 7.6: Important Modules in AspectJ

- Package *org.aspectj.lang.reflect*: Contains types that provide additional information about each possible join point signature.
- Packages *org.aspectj.runtimeorg.aspectj.runtime.internal*, and *org.aspectj.runtime.cflowstack*: Allow to handle the control flow pointcut characteristics and to define all the corresponding objects such as the counter and the stack, which allow the implementation of the control flow pointcut. Furthermore, *org.aspectj.runtime.internal* contains a class named *AroundClosure.java*, which stores the states of join pointcuts when the control flow pointcut instruction `proceed` is executed.
- Package *org.aspectj.runtime.reflect*: Contains classes that implement the different interfaces of the package *org.aspectj.lang.reflect*.

### 7.3.2 Weaver packages

We present hereafter the different packages in the module *weaver*:

- Package *org.aspectj.weaver*: Is the most important package in the `ajc` compiler implementation. It provides classes that represent AO objects like: *Advice.java*, *Shadow.java*, *ShadowMunger.java*, *JoinPointSignature.java*, etc., and other classes that allow the weaving.
- Package *org.aspectj.weaver.ast*: Contains classes that visit different AST pointcut nodes in order to find the residues when matching join points.
- Package *org.aspectj.weaver.bcel*: Provides classes that implement the bytecode injection during advice weaving on the different bytecodes.
- Package *org.aspectj.weaver.internal.tools*: Provides classes that implement from *org.aspect-j.weaver.tools* interfaces.
- Packages *org.aspectj.weaver.loadtime* and org.aspectj.weaver.ltw: Contain classes that replace standard loading mechanism by a different implementation that takes into account the weaving context.
- Package *org.aspectj.weaver.patterns*: Allows to construct objects from patterns. For example, the classe *PointcutParser* in this package creates objects for pointcut patterns as exemplified in Figure 7.4.
- Package *org.aspectj.weaver.reflect*: Is a reflection API that contains classes that are used for purposes of resolution based on the class path *java.lang.reflect*.
- Package *org.aspectj.weaver.tools*: Provides a set of interfaces for third-parties wishing to integrate AspectJ weaving capabilities into their environments [79].

## 7.4 Implementation

This section presents the AspectJ extension that we implemented to handle the new pointcuts: `getLocal`, `setLocal`, and `dflow`. We will first present what we added to the `ajc` compiler to take into account the local variables pointcuts. In a second step, we will describe the new package, *org.aspectJ.weaver.dataflow*, that we implemented for the data flow pointcut and finally, we will give a summarizing example.

### 7.4.1 Local Variable Pointcuts Implementation

In order to incorporate the new pointcuts: `getLocal` and `setLocal` in the `ajc` implementation, we considered the different parts of the back-end compiler. We present hereafter the modifications that we did on each part.

- **Parsing:**

  In this part, we had to:

  - Add methods to the class *PatternParser.java* in the package *org.aspectj.weaver.patterns* to recognize the new pattern *LocalVariablePatternList* that represents the list of targeted local variables.
  - Add in the same package *org.aspectj.weaver.patterns* a new class called *LVSignaturePattern.java*, which allows saving information about the method and local variable signature patterns.
  - Define a new constructor in the class *KindedPointcut.java* in the package *org.aspectj.weaver.patterns* that defines the *getLocal* and *setLocal* pointcuts.
  - Enrich the class *Member.java* in the package *org.aspectj.weaver* with two new member kinds *setLocalField* and *getLocalField*

- **Creating Shadow Mungers:**

  We did not change this part since the process of shadow munger creation remains the same.

- **Making Shadows:**

  Concerning this phase, we worked essentially in the package *org.aspectj.weaver.bcel*. This package contains classes that handle bytecode instructions. We enriched the class *BcelWorld.java*, *BcelShadow.java* and *BcelClassWeaver.java* to allow setting the correspondence between the two new join points and the related bytecode instructions.

- **Matching:**

  For this part, we enriched the class *BcelClassWeaver.java* with two new methods *matchSetLocal* and *matchGetLocal* that allow matching get and set local variables join points against the existing shadow mungers.

- **Weaving:**

  The last part of the back-end compiler remains unchanged since AspectJ defines a weaving process, which suits our implementation.

### 7.4.2 Data Flow Pointcut Implementation

The implementation of the `dflow` pointcut is inspired from the well-known def-use chains mechanism in data flow analysis. In this section, first we present the design behind the implementation and then we discuss the implementation details.

**Design**

A data flow analysis on the bytecode needs to follow all the read/write operations performed on the operand stack, the local variables table, or the constant pool. We have envisaged first to perform a dynamic analysis. In this case, the idea was to store the history of all the read/write operations done on these different structures. This would imply to inject bytecode instructions before each bytecode instruction that manipulates the operand stack, the local variables table, or the constant pool. Afterwards, we preferred a static analysis because of the costly overhead induced by a dynamic analysis. In fact, the initial program would become very heavy with a dynamic analysis and we would have a significant impact on the running time.

In the proposed implementation, we statically set dependencies between the different bytecode instructions in a method. A bytecode $b$ depends on another bytecode $b'$ if and only if $b$ is using a value that has been defined by $b'$. We will use the notation $\mathcal{D}ep(b)$ to represent the set of bytecode instructions on which depends $b$. The relation $\mathcal{D}ep$ defined between two bytecode instructions $b$ and $b'$ is transitive. Hence, we can reformulate the rule that joins together a join point with a data flow pointcut as a relation between bytecode instructions as follows:

*A bytecode b matches a data flow pointcut* `dflow` *(p) if and only if it exists another bytecode* $b'$ *that matches p and* $b' \in \mathcal{D}ep(b)$.

Notice that the current implementation of the `dflow` pointcut performs an intra-procedural analysis and considers only the case where the joinpoints are single bytecodes.

**Implementation**

We describe in the following the extension that we implemented in the `ajc` compiler in order to handle the `dflow` pointcut. First, we added a new class called *DFlowPointcut.java* to the package *org.aspectj.weaver.patterns* that defines data flow pointcuts.

In the "Parsing", part we enriched the class *PatternParser.java* to represent a data flow pointcut as an instance of the class *DflowPointcut*.

The part "Creating Shadow Mungers" is unchanged in this case as for the case of the local variables pointcuts since the process of creating shadow mungers remains the same.

For the other parts of the back-end compiler, we created a new package in the weaver module called *org.aspectJ.weaver.dataflow* and we extended also the class *BcelClassWeaver.java*.

The new package, as described in Figure 7.7, contains four classes: *Dependencies.java*, *ExecutionVisitor.java*, *InstructionTag.java*, and *MethodToDependencies.java*. In the following, we summarize the aim of each of theses classes and present the extensions performed on the class *BcelClassWeaver.java*.

**InstructionTag.java**

In order to differentiate between different occurrences of a same bytecode, the class *InstructionTag.java* represents a bytecode instruction as a combination of the bytecode itself and its line number.

**Dependancies.java**

This class allows to represent the dependencies of an instruction bytecode. It associates for a given instruction the list of instructions on which it depends. The class *Dependancies.java* is shown in Figure 7.8.

**ExecutionVisitor.java**

This class implements the interface *Visitor* defined in the package *org.aspectj.apache.bce-l.generic* of *ajc*. This interface implements the visitor pattern programming style. The class *ExecutionVisitor.java* that implements this interface can handle all types of instructions with the properly methods just by calling the *accept()* method.

In *ExecutionVisitor.java*, we use a stack of instruction tags called *lastDefiningInstructions*, which contains all the instruction tags that have pushed values in the operand stack. The instruction tags are popped from the stack *lastDefiningInstructions* when the values that they push on the operand stack are popped by another visited instruction.

For a better understanding, we describe the visit methods in the case of `aload`, `astore`, and `iadd` instructions.

- Instruction `aload`: The method in *ExecutionVisitor.java* that visits the `aload` instruction is shown in Figure 7.9. When visiting an `aload` instruction, the instruction is pushed onto the stack *lastDefiningInstructions* because this instruction puts a value onto the operand stack. The dependencies of the current

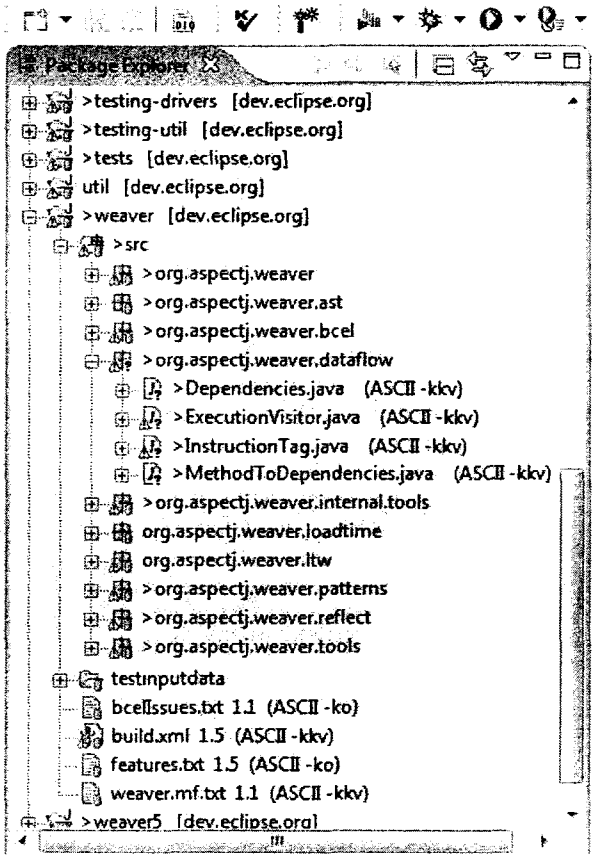



Figure 7.7: Data flow Package

`aload` instruction are either empty or contains a branch instruction. The flag *branchFlag* indicates if the `aload` instruction number line was the target of a branch instruction.

- Instruction `astore`: The method in *ExecutionVisitor.java* that visits the `astore` instruction is shown in Figure 7.10. The current `astore` instruction depends on the last instruction *i* that has been pushed onto the stack *lastDefiningInstructions* because it will pop from the operand stack the value pushed by *i*. Furthermore, if the current instruction line number is target of a branch instruction, this branch instructions is added to the `astore` instruction dependencies.

- Instruction `iadd`: The method in *ExecutionVisitor.java* that visits the `iadd` instruction is shown in Figure 7.11. Contrarily to the `aload` and `astore`, the `iadd` pops and pushes values from the operand stack. Indeed, the `iadd` instruction pops two values from the operand stack, performs the addition of the

```
package org.aspectj.weaver.dataflow;

import java.util.List;
import org.aspectj.apache.bcel.generic.MethodGen;

public class Dependencies{

  private MethodGen method;
  private InstructionTag instruction;
  private List<InstructionTag> list;

  public Dependencies(MethodGen method,InstructionTag instruction, List<InstructionTag>
      list ){
    this.method = method;
    this.instruction = instruction;
    this.list = list;
  }

  public MethodGen getMethod (){
    return method;
  }

  public InstructionTag getInstruction(){

    return instruction;
  }

  public List<InstructionTag> getDependanciesList(){

    return list;
  }

  public String toString(){
    return "( "+method.toString()+" "+instruction.toString()+": "+list.toString()+" )";
  }
}
```

Figure 7.8: Dependencies Class

two values and then pushes the result of the addition onto the operand stack. For this reason, when an `iadd` instruction is visited, its dependencies are set to the two last instructions pushed onto the stack *lastDefiningInstructions*. If the `iadd` instruction line number is the target of a branch instruction, this branch instructions is added also to the dependencies. Besides, the stack *lastDefiningInstructions* is updated. In fact, the two last instructions that have been pushed are popped and the current instruction `iadd` is pushed.

**MethodToDependencies.java**

This class allows to build the dependencies between the different bytecode instructions. A method called *execute()*, shown in Figure 7.12, will initiate visitors for the bytecode instructions that will build the dependencies for each bytecode instruction.

Besides, the method *execute()* sets the value of the branch flag that is needed during the execution of

```
public void visitALOAD(ALOAD o){
  int linenumber = getLineNumber(passage);
  passage++;
  InstructionTag it = new InstructionTag(linenumber,o);
  if(branchFlag){
    List<InstructionTag> l = new ArrayList<InstructionTag>();
    InstructionTag tag = target.get(actualposition).getInitialInstruction();
    l.add(0,tag);
    Dependencies dep = new Dependencies(currentmethod,it,l);
    dependencies.put(counter,dep);
    counter++;
    branchflag = false;
  }

  int size = lastdefininginstructions.size();
  lastdefininginstructions.add(size,it);

}
```

Figure 7.9: Visit Method for `aload`

```
public void visitASTORE(ASTORE o){
    int linenumber = getLineNumber(passage);
    InstructionTag it = new InstructionTag(linenumber,o);
    passage++;
    List<InstructionTag> l = new ArrayList<InstructionTag>();
    int size = lastdefininginstructions.size();
    l.add(0,lastdefininginstructions.get(size-1));
    if(branchFlag){
      InstructionTag tag = target.get(actualposition).getInitialInstruction();
      l.add(1,tag);
      branchFlag = false;
    }
    Dependencies dep = new Dependencies(currentmethod,it,l);
    dependencies.put(counter,dep);
    lastdefininginstructions.remove(size-1);
    counter++;
}
```

Figure 7.10: Visit Method for `astore`

each visit method as shown in Figure 7.9, Figure 7.10, and Figure 7.11. Such a flag allows to add a branch instruction tag to the visited bytecode instruction dependencies in case where this latter is the target of the branch instruction. All the calculated dependencies are stored in a table defined in the class ExecutionVisitor.java.

**Extension of *BcelClassWeaver.java***

The mechanism of matching bytecode instructions against pointcuts is handled in the class *BcelClassWeaver.java* in a method called *match*. In order to implement data flow pointcuts matching, we first

```
public void visitIADD(IADD o){
    int linenumber = getLineNumber(passage);
    InstructionTag it = new InstructionTag(linenumber,o);
    passage++;
    List<InstructionTag> l = new ArrayList<InstructionTag>();
    int size = lastdefininginstructions.size();
    l.add(0,lastdefininginstructions.get(size-2));
    l.add(1,lastdefininginstructions.get(size-1));
    if(branchFlag){
      InstructionTag tag = target.get(actualposition).getInitialInstruction();
      l.add(2,tag);
      branchFlag = false;
    }
    Dependencies dep = new Dependencies(currentmethod,it,l);
    dependencies.put(counter,dep);
    lastdefininginstructions.remove(size-1);
    lastdefininginstructions.remove(size-2);
    lastdefininginstructions.add(size-2,it);
    counter++;
}
```

Figure 7.11: Visit Method for `iadd`

added methods to the class *BcelClassWeaver.java* that propagate the data flow dependencies for each bytecode instruction. Secondly we extended the *match* method of *BcelClassWeaver.java* to perform the matching upon the built dependencies. Notice that the most important phase in the implementation is to collect the data flow dependencies of each bytecode instruction.

### 7.4.3 Example

We present in Figure 7.13 a snapshot of an AspectJ example containing the new pointcuts that we have implemented. In the example, the aspect contains a before advice that logs all the join points that set a public field, which is in the data flow of a private field. We can remark that one join point has been matched by the pointcut. Indeed, the instruction "*publicInfo=localstr*" is an instruction in which the public field *publicInfo* is set to a value that originates from the private field *sensitiveInfo*.

```
package org.aspectj.weaver.dataflow;
...

public class MethodToDependencies{
  private MethodGen mg;
  private InstructionHandle[] ihs;
  private ConstantPoolGen cpg;
  private Frame frame;
  private List<Target>  target = new ArrayList<Target>();
  private List<Integer> linenumbers = new ArrayList<Integer>();

...

  public void execute(){
    ExecutionVisitor ev = new ExecutionVisitor();
    ev.setConstantPoolGen(cpg);
    ev.setFrame(frame);
    ev.setLineNumbers(linenumbers);
    ev.setTargets(target);
    for(int i = 0; i < ihs.length;i++){
      //check whether current instruction handle is a branch instruction target or not
      int cpt = 0;
      for(Iterator j = target.iterator();j.hasNext();){
        Target t = (Target)j.next();
        InstructionTag tartag = t.getTarget();
        Instruction tarins = tartag.getInstruction();
        int pos = tartag.getLineNumber();
        if(ihs[i].getInstruction().equals(tarins)
          && ihs[i].getPosition()== pos){
          ExecutionVisitor.setBranchFlag(true);
          ExecutionVisitor.setActualTargetPosition(cpt);
        }
        cpt++;
      }

      ihs[i].accept(ev);
    }
  }
...

}
```

Figure 7.12: Method *execute()* in *MethodtoDependencies.java*

# 7.5 Conclusion

This chapter presents the extension implemented on the `ajc` compiler version 1.5. This extension allows to handle interesting pointcuts from security pointcuts, namely `setLocal`, `getLocal`, and `dflow`. These new pointcuts allow to perform data flow analysis and to avoid sensitive data disclosure. Hence, an analysis of the dependency relationships has been implemented to track data dependencies between instructions. This analysis takes into consideration transitivity relationships and branch instructions.


Figure 7.13: Screenshot for Implemented Pointcuts

# Chapter 8

# Conclusion

Computers have become an integral part of everyday life and the last decade has seen a rapid grow of computation and communication in critical infrastructures such as financial, telecommunication, energy, and transportation. As fast as the computing infrastructure is growing, computer security attacks at all levels are growing faster implying an acute need to develop tools that better prevent attacks. As a consequence, application security hardening becomes a priority in the IT market today. Aspect Oriented Programming is considered as a good candidate for application security hardening as it automates the weaving process. This research provides practical and theoretical solutions for security hardening of Java applications using AspectJ. We restate in this chapter the key technical contributions of our work and present the possible extensions. These contributions fall in two main categories:

- *At the practical level:* We provided [5] the design and the implementation of new AspectJ pointcuts: `getLocal`, `setLocal`, and `dflow`. The pointcuts `getLocal` and `setLocal` target program locations where local variables are respectively read or set whereas the pointcut `dflow` is related to information flow. We have shown, through AspectJ security assessment, the usefulness of those pointcuts. The extension that we developed has been implemented on the open source AspectJ compiler `ajc` version 1.5 that is available on the Eclipse site. Our extension to AspectJ is the first extension that integrates pointcuts targeting local variable accesses and data flow information flow. Previous contributions have shown the usefulness of the data flow pointcut, but none of them has implemented it.
- *At the theoretical side:* Our contribution is three-fold: First, we have provided a dynamic semantics for the JVML [11] because the weaving in AspectJ combines the JVML representation of the initial

program and the enriched JVML representation of the aspects producing a woven JVML program. Accordingly, we explored the completeness of the most prominent proposals advanced in the literature and we proposed [1,2] an operational semantics that includes features that have not been addressed in previous contributions. Hence, the presented semantics is, to our best knowledge, the first semantics that deals with multi-threading, synchronization, method exception, method invocation, exception handling, object creation, object's fields manipulation, stack manipulation, local variable access, modifiers, etc. Second, after studying the informal official specification of AspectJ as published by Eclipse, we have proposed a small step dynamic operational semantics that covers a large subset of AspectJ [9,10]. We formalized AspectJ features such as the residue generation and we described how the dynamic tests are generated in the case of dynamic pointcuts. Hence, this semantics compiles the know-how of the AspectJ community into an elegant framework. It is also, to our best knowledge, the first formal semantics for AspectJ. Third, after a security assessment of AspectJ, we elaborated a security aspect-oriented calculus called λ_SAOP based on the well-known λ_calculus together with its semantic foundations [3,4]. We have extended and accommodated the effect-based inference algorithm to take matching and weaving processes into consideration. In addition, we established the required soundness and preservation proofs. This contribution is useful to the users/researchers in the field of Aspect Oriented Programming and those who are interested in solving security problems.

**Future Work**

We present now the future work regarding the different aspects of the research presented in this thesis. The following continuations can be considered:

- Design and implementation of an interprocedural data flow analysis. The resulting interprocedural data flow analysis will be compatible with the current intraprocedural data flow implementation and handle a whole program analysis whereas the current data flow analysis is a per function analysis.
- Extension of the AspectJ semantics presented in Chapter 5 with features that we have not handle. For instance, adding the around advice semantics and the cflow semantics.
- Implementation in AspectJ of other useful features enumerated in Chapter 3. This will ensure a better security hardening of applications. For instance, implementing a loop pointcut will prevent some denial of service attacks.
- Investigation for more case studies related to security problems.

Finally, the future work will pave the way for a fully-fledged framework for aspect oriented security hardening.

# Bibliography


[1] D. Alhadidi, N. Belblidia, and M. Debbabi. AspectJ and Security. In *PST '06: Proceeding of the International Conference on Privacy, Security and Trust, Ontario, Canada*, October 2006.

[2] D. Alhadidi, N. Belblidia, and M. Debbabi. AspectJ Assessment from a Security Perspective. In *PTITS'2006: Proceedings of the Workshop on Practice and Theory of IT Security, Ontario, Canada*, May 2006.

[3] D. Alhadidi, N. Belblidia, M. Debbabi, and P. Bhattacharya. An AOP Extended Lambda-Calculus. In *SEFM'07, 5th IEEE International Conference on Software Engineering and Formal Methods, London, UK*, pages 183–194. IEEE Computer Society, 2007.

[4] D. Alhadidi, N. Belblidia, M. Debbabi, and P. Bhattacharya. λ_SAOP: A Security AOP Calculus. *To Appear in Computer Journal*, 2008.

[5] D. Alhadidi, A. Boukhetouta, N. Belblidia, M. Debbabi, and P. Bhattacharya. The Data Flow Pointcut - A Formal and Practical Framework. In *AOSD'09, 8th International Conference on Aspect-Oriented Software Development, Charlottesville, Virginia, USA, (to appear)*. ACM, 2009.

[6] J. Alves-Foss and F.S. Lam. Dynamic Denotational Semantics of Java. In *Formal Syntax and Semantics of Java*, pages 201–240, 1999.

[7] J. H. Andrews. Process Algebraic Foundations of Aspect Oriented Programming. In *Reflection*, pages 187–209, 2001.

[8] H. Barendregt and K. Hemerik. Types in Lambda Calculi and Programming Languages. In *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark*, pages 1–35, 1990.

[9] N. Belblidia and M. Debbabi. Formalizing AspectJ Weaving for Static Pointcuts. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods,Pune, India*, pages 50–59. IEEE Computer Society, 2006.

[10] N. Belblidia and M. Debbabi. Towards a Formal Semantics for AspectJ Weaving. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2008, Oxford, UK*, volume 4228/2006 of *Lecture Notes in Computer Science,*, pages 155–171. Springer Berlin / Heidelberg, 2006.

[11] N. Belblidia and M. Debbabi. A Dynamic Operational Semantics for JVML. In *Journal of Object Technology*, volume 6, pages 71–100. 2007.

[12] P. Bertelsen. Semantics of Java Bytecode. Technical report, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Copenhagen, Denmark, 1997.

[13] P. Bertelsen. Dynamic Semantics of Java Bytecode. *Future Genration Computer systems*, 16(7):841–850, 2000.

[14] G. Bigliardi and C. Laneve. A Type System for JVM Threads. Technical Report UBLCS-2000-06, Department of Computer Science, University of Bologna, 2000.

[15] R. Bodkin. Enterprize Security Aspects. In *AOSD'04 Workshop: International Conference on Aspect-Oriented Software Development.*

[16] K. Böllert. On Weaving Aspects. In *ECOOP'99, the 13th European Conference on Object-Oriented Programming, Lisbon, Portugal*, 1999.

[17] E. Borger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *Mathematical Foundations of Computer Science*, pages 17–35, 1998.

[18] E. Borger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 353–404. LNCS Springer, 1999.

[19] J. Borner. Semantics for a Synchronized Block Join Point. http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-join-point/, July 2005. Last Visited: November 2008.

[20] P. Cenciarelli. Towards a Modular Denotational Semantics of Java. In *Proceedings of the Workshop on Object-Oriented Technology*, page 105, London, UK, 1999. Springer-Verlag.

[21] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Operational Semantics of Multi-Threaded Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 157–200. LNCS Springer, 1999.

[22] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, pages 345–363, 1936.

[23] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[24] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of Foundations of software Engineering*, Vienne, Austria, 2001.

[25] R. M. Cohen. The Defensive Java Virtual Machine Specification. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

[26] H. Curry. Functionality in combinatory logic. In *"Proceedings of Natural Academy of Sciences U.S.A., USA"*, volume 20, pages 584–590, 1934.

[27] M. Debbabi, Z. Aidoud, and A. Faour. On the Inference of Structured Recursive Effects with Subtyping. *Journal of Functional and Logic Programming*, 1997(5), 1997.

[28] Demeter Group. Demeter: Aspect-Oriented Software Development. Available at `http://www.ccs.neu.edu/research/demeter/`. Last Visited: November 2008.

[29] B. DeWin. AOSD is an Enabler for Good Enough Security. Available at `http://citeseer.ist.psu.edu/728786.html`, 2003. Last Visited: November 2008.

[30] B. DeWin. Engineering Application Level Security through Aspect Oriented Software Development. Available at `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/resources/publications/41140.pdf`, 2004. Last Visited: November 2008.

[31] B. DeWin, F. Piessens, W. Joosen, and T. Verhanneman. On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering. Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, 2002, Applied Computer Security Associates (ACSA) available at `http://www.acsac.org/waepssd/index.html`, 2002.

[32] B. DeWin, B. Vanhaute, and B. De Decker. Security Through Aspect-Oriented Programming. In B. De Decker, F. Piessens, J. Smits, and Van Herreweghen, editors, *Advances in Network and Distributed Systems Security*, pages 125–138, 2001.

[33] B. DeWin, B. Vanhaute, and B. De Decker. How Aspect-Oriented Programming Can Help to Build Secure Software. *Informatica*, 26(2):141–149, 2002.

[34] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[35] R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[36] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java Type Soundness Revisited. Technical report, Imperial College London, 2000. Available at `citeseer.ist.psu.edu/article/drossopoulou00java.html`.

[37] C. Dutchyn, G. Kiczales, and H. Masuhara. Aspect Sand Box Project. Available at `http://www.cs.ubc.ca/labs/spl/projects/asb.html`, 2002. Last Visited: November 2008.

[38] Eclipse. The Eclipse AspectJ Implementation. Available at `http://www.eclipse.org/aspectj`. Last Visited: November 2008.

[39] S. N. Freund and J. C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *Proc. 14th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, volume 34(10). ACM Press, 1999.

[40] S. N. Freund and J. C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[41] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, page 271Ũ321, December 2003.

[42] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *LISP and Functional Programming*, pages 28–38, 1986.

[43] J.Y. GIRARD. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[44] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A Metalanguage for Interactive Proof in LCF. pages 119–130, Tucson, Arizona, January 1978.

[45] M. Hagiya and A. Tozawa. On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines. In *SIG-Notes, PRO-17-3*, pages 13–18. Information Processing Society of Japan, 1998.

[46] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, March 2005.

[47] K. Havelund and K. G. Larsen. The Fork Calculus. In *ICALP'93, Automata, Languages and Programming, 20nd International Colloquium, Lund, Sweden*, pages 544–557, 1993.

[48] M. Huang, C. Wang, and L. Zhang. Toward a Reusable and Generic Security Aspect Library. In *AOSDSEC'04: AOSD Technology for Application-level Security*, Lancaster, London, March 2004.

[49] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Technical Report YALEU/DCS/RR777, Yale University, 1991.

[50] B. Jacobs. A Formalization of Java's Exception Mecahnism. In David Sands, editor, *Programming languages and systems*, pages 284–301. LNCS Springer-Verlag, 2001.

[51] B. Jacobs and E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, 2003.

[52] R. Jagadeesan, A. Jeffrey, and J. Riely. A Calculus of Untyped Aspect-Oriented Programs. In *ECOOP'03*, pages 54–73, Darmstadt , ALLEMAGNE, 2003.

[53] K. Kawauchi and H. Masuhara. Dataflow Pointcut for Integrity Concerns. AOSD Technology for Application-level Security Workshop, March 23, Lancaster, UK, 2004.

[54] G. Kiczales. The Fun has Just Begun. AOSD'03 Keynote, available from `http:www.cs.ubc.ca/~gregor`, 2003.

[55] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J.Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[56] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[57] H. Kim. AspectC#: An AOSD Implementation for C#. Technical Report TCD -CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[58] G. Klein and M. Wildmoser. Verified Bytecode Subroutines. *Journal of Automated Reasoning*, 30(3–4):363–398, 2003.

[59] Cigital Labs. An Aspect-Oriented Security Assurance Solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, Dulles, Virginia, USA, 2003.

[60] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User Authentication and Authorization in the Java Platform. In *15th Annual Computer Security Applications Conference*, pages 285–290. IEEE Computer Society Press, 1999.

[61] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, 2002. ACM Press.

[62] C. Laneve. A Type System for Jvm Threads. *Theoretical Computer Science*, 290((1)):741 – 778, 2003.

[63] X. Leroy and P. Weis. Polymorphic Type Inference and Assignment. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 291–302. ACM Press, 1991.

[64] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

[65] J.M Lucassen. *Types and Effects Towards the Integration of Functional and Imperative Programming*. PhD thesis, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1987.

[66] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *APLAS*, pages 105–121, 2003.

[67] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *ECOOP 2003*, 2003.

[68] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002*, number 02-06 in Technical Report, pages 17–26. Department of Computer Science, Iowa State University, 2002.

[69] G. McGraw and E.Felten. *Securing Java Getting Down to Business with Mobile Code*. John Wiley & Sons, 1999.

[70] Sun Microsystems. Java cryptography extension. Available at url-http://java.sun.com/products/jce/index.html. Last Visited: November 2008.

[71] A.C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[72] H. R. Nielson and F. Nielson. *Semantics with Applications: a Formal Introduction*. John Wiley & Sons, Inc., 1992.

[73] T. Nipkov and D. V. Oheimb. Machine Checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS Springer, 1999.

[74] D.V. Oheimb. Axiomatic Semantics for Java$^{light}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000.

[75] D. Orleans and K. Lieberherr. Adaptive Programming with Traversals and Visitors. Available at `http:/www.ccs.neu.edu/research/demeter/posters/introDemeterJava`, 1997. Last Visited: November 2008.

[76] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. Tech Report NU-CCS-2001-02, Northeastern University, Boston, MA 02115, USA, 2001.

[77] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. Technical Report NU-CCS-2001-02, Northeastern University, Boston, MA 02115, USA, 2002.

[78] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[79] Palo Alto Research Center. The Aspectj Programming Guide. Available at `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html`. Last Visited: November 2008.

[80] Palo Alto Research Center. The Aspectj Programming Guide: AspectJ Langage Semantics: Join Points. Available at `http://www.eclipse.org/aspectj/doc/released/progguide/semantics.html`. Last Visited: November 2008.

[81] D. L Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[82] J. C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings, Colloque sur la Programmation, Paris.*

[83] V. Shah and F. Hill. Using Aspect-Oriented Programming for Addressing Security Concerns. In *ISSRE2002*, pages 115–119, 2002.

[84] I. Siveroni. Operational Semantics of the Java Card Virtual Machine, 2004. J. Logic and Automated Reasoning, 2004. To appear.

[85] I. Siveroni and C. Hankin. A Proposal for the JCVMLe Operational Semantics. citeseer.ist.psu.edu/siveroni01proposal.html, 2001. Last Visited: November 2008.

[86] O. Spinczyk, A. Gal, and W. chröder Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia, 2002.

[87] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.

[88] D. Syme. Proving Java Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 83–118. LNCS Springer, 1999.

[89] J.P. Talpin and P. Jouvelot. Polymorphic Type Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[90] J.P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[91] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 2000.

[92] P. Tarr, H. Ossher, and S. M. Sutton. Hyper/j: Multi-dimensional separation of concerns for java. Available at `http://trese.cs.utwente.nl/aosd2002/index.php?content=hyperj"`, 2002. Last Visited: November 2008.

[93] S.T Teoh, T.J. Jankun-Kelly, K.L Ma, and F.S. Wu. Visual data analysis for detecting flaws and intruders in computer network systems. *IEEE Computer Graphics and Applications, special issue on Visual Analytics*, 2004.

[94] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, UK, May 1988.

[95] B. Vanhaute and B. DeWin. AOP, Security and Genericity. 1st Belgian AOSD Workshop, Vrije Universiteit Brussel, Brussels, Belgium, November 8, 2001, 2001.

[96] J. Viega, J. T. Bloch, and C. Pravir. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, 2001.

[97] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, 2003.

[98] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[99] Bart De Win, Bart De Win, Bart De Decker, Bart De Decker, Bart Vanhaute, Bart Vanhaute, Bart Vanhaute, Bart De, Win Bart, and De Decker. Building frameworks in aspectj. In *Workshop on Advanced Separation of Concerns (ECOOP 2001*, pages 1–6, 2001.

[100] A. K. Wright. Typing References by Effect Inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

# Appendices

## Appendix I: JVML Semantics Utility Functions

1. The function accessAllowedF returns true if a given field is visible from a given method:

   accessAllowedF : $Method \times Field \rightarrow Boolean$

   accessAllowedF($m$,$f$) = ( $\neg$ isPrivateF($f$) $\vee$ ($f$.*fromClass*=$m$.*fromClass*))

2. The function accessAllowedM returns true if a given method is visible from another given method:

   accessAllowedM : $Method \times Method \rightarrow Boolean$

   accessAllowedM($m$,$m'$) = ( $\neg$ isPrivateM($m'$) $\vee$ ($m'$.*fromClass*=$m$.*fromClass*))

3. The function activateThreads returns a new Java stack where all the threads waiting for objects or classes in a given list are activated:

   activateThreads : $JavaStack \times Store \times environment \times$ (*ClassOrLocation*) `-list` $\rightarrow$ *JavaStack*

   activateThreads($\mathcal{JS}$, $S$,$\mathcal{JE}$, $l$) = $\mathcal{JS}'$ if

   $$\begin{cases} \mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in Dom(\mathcal{JS}) \wedge i \notin \text{waitingThreads}(S, \mathcal{JE}, l) \\ \mathcal{JS}'(i) = \text{active}(\mathcal{JS}, i), \forall i \in Dom(\mathcal{JS}) \wedge i \in \text{waitingThreads}(S, \mathcal{JE}, l) \end{cases}$$

4. The function active returns the thread information of a given entry $i$ in a given Java stack with a state equal to `active`:

   active : $JavaStack \times Nat \rightarrow ThreadInformation \times State$

   active($\mathcal{JS}$,$i$) = ($\mathcal{JS}(i)$.*threadInformation*,`active`)

5. The function addToClassWaitingList adds a given thread in the waiting list of a given class:

   addToClassWaitingList: *Environment*×*ClassType*×*ThreadId* → *Class*

   addToClassWaitingList($\mathcal{JE}$,*ct*,*id*) = $\mathcal{JE}$(*ct*)[*monitorClass.waitList* ← id::$\mathcal{JE}$(*ct*).*monitorClass.waitList*]

6. The function addToObjectWaitingList adds a given thread in the waiting list of a given object:

   addToObjectWaitingList: *Store*×*Location*×*ThreadId* → *JavaObject*

   addToObjectWaitingList(*S*,*Loc*,*id*) =*S*(*Loc*)[*monitor.waitList* ← id::*S*(*loc*).*monitor.waitList*]

7. The function allInterfaces gives all the interfaces of a given set of classes.

   allInterfaces : *Environment*×*(RefOrNoneType)*-`set` → *(ReferenceType)*-`set`

   allInterfaces($\mathcal{JE}$,) =

   allInterfaces($\mathcal{JE}$,{`Object`})=

   allInterfaces($\mathcal{JE}$,{`None`})=

   allInterfaces($\mathcal{JE}$,{*ct*}) = $\mathcal{JE}$(*ct*).*interfaces* ∪ allInterfaces($\mathcal{JE}$,{$\mathcal{JE}$(*ct*).*superClass*})

   ∪ allSuperClasses($\mathcal{JE}$,$\mathcal{JE}$(*ct*).*interfaces*) if *ct* ≠ `Object` ∧ *ct* ≠ `None`

   allInterfaces($\mathcal{JE}$,{*ct*} ∪ *ctSet*) = allInterfaces($\mathcal{JE}$,{*ct*}) ∪ allInterfaces($\mathcal{JE}$,*ctSet*) if *ctSet* ≠

8. The function allInstanceFields returns the set of all the instance fields, included the inherited fields, of a given class. Static fields are not returned by this function:

   allInstanceFields : *Environement*×*ReferenceType* → (*Field*)-set

   allInstanceFields($\mathcal{JE}$, "`Object`") = ∅

   allInstanceFields($\mathcal{JE}$, *ct*) = setOf($\mathcal{JE}$(*ct*).*fields*) ∪ allInstanceFields($\mathcal{JE}$(*ct*).*superClass*)

   if *ct* ≠ "`Object`"

9. The function allSuperClasses gives all the super classes of a given set of classes.

allSuperClasses : *Environment*×*(RefOrNoneType)*-set → *(ReferenceType)*-set

allSuperClasses($\mathcal{JE}$,)=

allSuperClasses($\mathcal{JE}$,{Object}) =

allSuperClasses($\mathcal{JE}$,{None}) =

allSuperClasses($\mathcal{JE}$,{*ct*}) = $\mathcal{JE}$(*ct*).*superClass* ∪ allSuperClasses($\mathcal{JE}$,{$\mathcal{JE}$(*ct*).*superClass*})

if *ct* ≠ Object ∧ *ct* ≠ None

allSuperClasses($\mathcal{JE}$,{*ct*} ∪ *ctSet*) = allSuperClasses($\mathcal{JE}$,{*ct*}) ∪ allSuperClasses($\mathcal{JE}$,*ctSet*)

if *ctSet* ≠

10. The function appropriatePcHandler returns the start address indicated by the first appropriate exception handler found given an environment, a program counter, an exception class type to catch and a list of exception handlers. If there is no appropriate exception handler in the list, the function return −1:

appropriatePcHandler: *Environment*×*Nat*×*ReferenceType*×*ExceptionTable*→ int

$$
\begin{aligned}
&\text{appropriatePcHandler}(\mathcal{JE}, pc, ct, [\,]) && = -1\\
&\text{appropriatePcHandler}(\mathcal{JE}, pc, ct, l) && = \text{head}(l).handler\\
& && \quad \text{if isAppropriateHandler}(\mathcal{JE}, pc, ct, \text{head}(l))\\
&\text{appropriatePcHandler}(\mathcal{JE}, pc, ct, l) && = \text{appropriatePcHandler}(\mathcal{JE}, pc, ct, \text{tail}(l))\\
& && \quad \text{if} \neg \text{isAppropriateHandler}(\mathcal{JE}, pc, ct, \text{head}(l))
\end{aligned}
$$

11. The function blockThreads sets the state of a thread to blocked in the Java stack:

blockThreads : *JavaStack*×*ThreadInformation* → *JavaStack*

$$
\text{blockThreads}(\mathcal{JS}, \mathcal{T}) = \mathcal{JS}' \text{ if } \begin{cases} \mathcal{T}.threadId = id \\ \mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in Dom(\mathcal{JS}) \wedge i \neq id \\ \mathcal{JS}'(id).state = \texttt{blocked} \\ \mathcal{JS}'(id).threadInformation = \mathcal{T} \end{cases}
$$

12. The function changeShadows takes a list of shadows, a program counter, and a natural number

and returns a new list of shadows. The returned list of shadows is similar to the given one except that the parts *start* and *end* of the shadows may be changed. In fact, if the part *start* or *end* of a shadow is greater than the given program counter, it is incremented with the given natural number:

changeShadows: (*Shadow*)-list×*Nat*×*Nat* → (*Shadow*)-list

changeShadows([],*pc*,*i*) = []

changeShadows(*msh*,*pc*,*i*) = head(*msh*)[*start*←*start*+*i*/*start* > *pc*, *end*←*end*+*i*/*end* > *pc*]::

changeShadows(tail(*msh*),*pc*,*i*)

13. The function changeThreads allows to change a given thread information to another given thread information in a given Java stack and maintaining the state to `active`:

changeThreads: *JavaStack*×*ThreadInformation*×*ThreadInformation* → *JavaStack*

$$\text{changeThreads}(\mathcal{JS}, \mathcal{T}, \mathcal{T}') = \mathcal{JS}' \text{ if } \begin{cases} \mathcal{T}.\text{threadId} = id \\ \mathcal{JS}'(i) = \mathcal{JS}(i), \forall i \in Dom(\mathcal{JS}) \wedge i \neq id \\ \mathcal{JS}'(id).\text{state} = \mathcal{JS}(id).\text{state} \\ \mathcal{JS}'(id).\text{threadInformation} = \mathcal{T}' \end{cases}$$

14. The function classMonitorEntered returns the same class than the given class identifier but containing the information that the class's monitor has been entered by a given thread:

classMonitorEntered : *Environment*×*ReferenceType*×*ThreadId* → *Class*

classMonitorEntered($\mathcal{JE}$,*ct*,*id*) = $\mathcal{JE}$(*ct*)[*monitorClass.waitList*←

suppress($\mathcal{JE}$(*ct*).*monitorClass.waitList*,id);

*monitorClass.threadOwner* ← *id*;*monitorClass.depth* ← $\mathcal{JE}$(*ct*).*monitorClass.depth*+1]

15. The function classMonitorExited returns the same class than the given class identifier but containing the information that the class's monitor has been exited by a given thread:

classMonitorExited : *Environment*×*ReferenceType*×*ThreadId* → *Class*

classMonitorExited($\mathcal{JE}$,*ct*,*id*)= $\mathcal{JE}$(*ct*)

[*monitorClass.threadOwner* ← "None"/ $\mathcal{JE}$(*ct*).*monitorClass.depth*=1;

*monitorClass.depth* ← *S*(*Loc*).*monitorClass.depth*-1]

16. The function default returns the default value of a given field:

    default : *Field* → *Value*

    default($f$) = 0 if typeOfField($f$)= *PrimitiveType*

    default($f$) = `Null` if typeOfField($f$)= *ReferenceType*

17. The function defaultFieldMap returns, given a list of fields, a map between those fields and their respective default values:

    defaultFieldMap : (*Field*)-set → (*Field* $\underset{m}{\rightarrow}$ *Value*)

    defaultFieldMap($l$) = $g$ if Dom($g$)=$l$ and g(f)=default(f) , $\forall$ f $\in$ $l$

18. The function depthClassLock returns the number of times a given class in a given environment has been reentered. If the object is not owned by any thread, the value returned is zero:

    depthClassLock : *Environment*×*ReferenceType* → *Nat*

    depthClassLock($\mathcal{JE}$,*loc*) = ($\mathcal{JE}$(*loc*).*monitorClass*).*depth*

19. The function depthLock returns the number of times an object in a given Location and a given store has been reentered. If the object is not owned by any thread, the value returned is zero:

    depthLock : *Store*×*Location* → *Nat*

    depthLock($S$,*loc*) = ($S$(*loc*).monitor).*depth*

20. The function dieThread returns a Java stack constructed by removing a given thread from a given Java stack:

    dieThread : *JavaStack*×*ThreadId* → *JavaStack*

    $$\text{dieThread}(\mathcal{JS}, id) = \mathcal{JS}' \text{ if } \begin{cases} Dom(\mathcal{JS}') = Dom(\mathcal{JS}) - \{id\} \\ \forall i \in Dom(\mathcal{JS}'); \mathcal{JS}'(i) = \mathcal{JS}(i) \end{cases}$$

21. The function getDynamicClass returns the dynamic class of an object at a given location in a given store:

    getDynamicClass : *Store*×*Location* → *ReferenceType*

    getDynamicClass($S$,*loc*) = $S$(*loc*).*classType*

22. The function getLocalValue returns an element at a given position in a given list:

getLocalValue : $(\tau)$-list $\times Nat \rightarrow \tau$

getLocalValue$(l,0)$ = head$(l)$

getLocalvalue$(l,i)$ = getLocalValue(tail$(l)$,$i$-1), $\forall\, i > 0$

23. The function getOneStackElem is identical to getLocalValue. It has been introduced only to let the comprehension of the semantic rules easier:

getOneStackElem : $(\tau)$-list $\times Nat \rightarrow \tau$

getOneStackElem$(l,i)$ = getLocalValue$(l,i)$

24. The function getStackElemts returns a sublist of a given list with a length equal to a given natural number:

getStackElemts : $(\tau)$-list$\times Nat \rightarrow$ $(\tau)$-list

getStackElemts$(l,0)$ = []

getStackElemts$(l,i)$ = head(l)::getStackElemts(tail$(l)$,$i$-1), $\forall\, i > 0$

25. The function head returns the first element in a given list:

head : $(\tau)$-list $\rightarrow \tau$

head$(v::l)$ = $v, \forall\ (v, l) \in \tau \times (\tau)$-list

26. The function ifThenElse returns, depending on the value of a given boolean value, the second or the third given argument:

ifThenElse : $Boolean \times \tau \times \tau \rightarrow \tau$

ifThenElse(`true`,$a$,$b$) = $a$

ifThenElse(`false`,$a$,$b$) = $b$

27. The function isAppropriateHandler returns true if a given exception handler is appropriate for a given program counter and a thrown exception in a specific environment otherwise the function returns false:

isAppropriateHandler: $Environment \times Nat \times ReferenceType \times ExceptionHandler \rightarrow Boolean$

isAppropriateHandler($\mathcal{JE}, pc, ct, h$) = `True` if

$$\begin{cases} h.\text{startPc} <= pc \\ h.\text{endPc} >= pc \\ h.\text{exceptionType} \in \text{allSuperClasses}(\mathcal{JE}, ct) \end{cases}$$

isAppropriateHandler($\mathcal{JE}, pc, ct, h$) = `false` otherwise.

28. The function isClassOwner returns true if in a given environment, a given thread Id is owner of a given class otherwise the function returns false:

    isClassOwner : *Environment*×*ReferenceType*×*ThreadId* → *Boolean*

    isClassOwner($\mathcal{JE}$,*c*,*id*) = (($\mathcal{JE}$(*c*).*monitorClass*).*threadOwner*= id)

29. The function isInitialized returns true if in a given environment, a given class is initialized otherwise the function returns false:

    isInitialized : *Environment* × *ReferenceType* → *Boolean*

    isInitialized($\mathcal{JE}$,*c*) = ($\mathcal{JE}$(*c*).*initialized*=1)

30. The function isInterface returns true if in a given environment, a given class is an interface otherwise the function returns false

    isInterface : *Environment* × *ReferenceType* → *Boolean*

    isInterface($\mathcal{JE}$,*c*) = ($\mathcal{JE}$(*c*).*interface*=1)

31. The function isLocked returns true if in a given store, a given location is locked otherwise the function returns false:

    isLocked : *Store*×*Location* → *Boolean*

    isLocked(*S*,*loc*) = (*S*(*loc*).*monitor*).*threadOwner* ≠ "`None`")

32. The function isMethodOf returns true if a class with an identifier belonging to a given list of class identifiers contains a method with the same signature than the given method signature otherwise the function returns false:

    isMethodOf : *Environment*×*MethodSignature*×*(ReferenceType)*-`set` → *Boolean*

isMethodOf($\mathcal{JE}$,*ms*,) = false

isMethodOf($\mathcal{JE}$,*ms*,{*ct*}) = true if *ms* $\in$ methodSignatures($\mathcal{JE}$(*ct*).*methods*)

isMethodOf($\mathcal{JE}$,*ms*,{*ct*}) = false if *ms* $\notin$ methodSignatures($\mathcal{JE}$(*ct*).*methods*)

isMethodOf($\mathcal{JE}$,*ms*,{*ct*} $\cup$ *ctSet*) = isMethodOf($\mathcal{JE}$,*ms*,{*ct*}) $\vee$ isMethodOf($\mathcal{JE}$,*ms*,*ctSet*)

33. The function isMethInterfaceResolved returns true if a given interface method signature is resolved for a given interface in a given environment otherwise the function returns `false`:[1]

isMethInterfaceResolved : *Environment*$\times$*MethodSignature*$\times$*ReferenceType* $\rightarrow$ *Boolean*

isMethInterfaceResolved($\mathcal{JE}$,*ms*,*ct*) = false if $\neg$isInterface($\mathcal{JE}$,*ct*)

$$\text{isMethInterfaceResolved}(\mathcal{JE},ms,ct) = \text{false if} \begin{cases} \text{isInterface}(\mathcal{JE},ct) \\ classes = \text{allInterfaces}(\mathcal{JE},\{ct\}) \cup \{ct,\texttt{"Object"}\} \\ \neg\text{isMethodOf}(\mathcal{JE},ms,classes) \end{cases}$$

$$\text{isMethInterfaceResolved}(\mathcal{JE},ms,ct) = \text{true if} \begin{cases} \text{isInterface}(\mathcal{JE},ct) \\ classes = \text{allInterfaces}(\mathcal{JE},\{ct\}) \cup \{ct,\texttt{"Object"}\} \\ \text{isMethodOf}(\mathcal{JE},ms,classes) \end{cases}$$

34. The function isMethResolved returns true if a given method signature is resolved for a given class in a given environment otherwise the function returns `false`:[2]

[1] An interface method *m* is resolved [64] for a given class *c* if *c* is an interface and *c* or the class "`Object`" or one of the interfaces of *C* declares *m*

[2] A method *m* is resolved for a class *c* [64] if *c* is not an interface and *c* , or one of the super classes of *c* or any of the superinterfaces of *c* declares *m*.

isMethResolved : $Environment \times MethodSignature \times ReferenceType \rightarrow Boolean$

isMethResolved($\mathcal{JE}$,$ms$,$ct$) = false if isInterface($\mathcal{JE}$,$ct$)

$$\text{isMethResolved}(\mathcal{JE},ms,ct) = \text{false if} \begin{cases} \neg\text{isInterface}(\mathcal{JE},ct) \\ classes1 = \{ct\} \cup \text{allSuperClasses}(\mathcal{JE},\{ct\}) \\ classes = classes1 \cup \text{allInterfaces}(\mathcal{JE},\{ct\}) \\ \neg\text{isMethodOf}(\mathcal{JE},ms,classes) \end{cases}$$

$$\text{isMethResolved}(\mathcal{JE},ms,ct) = \text{true if} \begin{cases} \neg\text{isInterface}(\mathcal{JE},ct) \\ classes1 = \{ct\} \cup \text{allSuperClasses}(\mathcal{JE},\{ct\}) \\ classes = classes1 \cup \text{allInterfaces}(\mathcal{JE},\{ct\}) \\ \text{isMethodOf}(\mathcal{JE},ms,classes) \end{cases}$$

35. The function isOwner returns true if an object in a given store and pointed by a given Location is acquired by a thread with a given Id otherwise the function returns false:

    isOwner : $Store \times Location \times ThreadId \rightarrow Boolean$

    isOwner($S$,$loc$,$id$) = (($S(loc).monitor$).$threadOwner = id$)

36. The function isClassLocked returns true if a class with a given class identifier is locked in a given environment otherwise the function returns false:

    isClassLocked : $Environment \times ReferenceType \rightarrow Boolean$

    isClassLocked($\mathcal{JE}$,$c$) = ($\mathcal{JE}(c).monitorClass.threadOwner \neq$ "None")

37. The function isPrivateF returns true if a given field is private otherwise the function returns false:

    isPrivateF : $Field \rightarrow Boolean$

    isPrivateF($f$) = (private $\in f.fieldModifiers$)

38. The function isPublicF returns true if a given field is public otherwise the function returns false:

    isPublicF : $Field \rightarrow Boolean$

    isPublicF($f$) = (public $\in f.fieldmMdifiers$)

39. The function isPrivateM returns true if a given method is private otherwise the function returns false:

isPrivateM : *Method* → *Boolean*

isPrivateM(*m*) = ( private ∈ *m.methodModifiers*)

40. The function isPublicM returns true if a given method is public otherwise the function returns false:

    isPublicM : *Method* → *Boolean*

    isPublicM(*m*) = ( public ∈ *m.methodModifiers*)

41. The function isStaticF returns true if a given field is static otherwise the function returns false:

    isStaticF : *Field* → *Boolean*

    isStaticF(*f*) = (static ∈ *f.fieldModifiers*)

42. The function isStaticM returns true if a given method is static otherwise the function returns false:

    isStaticM : *Method* → *Boolean*

    isStaticM(*m*) = (static ∈ *m.methodmMdifiers*)

43. The function isSynchronized returns true if a given method is synchronized otherwise the function returns false:

    isSynchronized : *Method* → *Boolean*

    isSynchronized(*m*) = (synchronized ∈ *m.methodModifiers*)

44. The function isThread returns true if a given class type is the class `Thread` or one of its subclasses otherwise the function returns false:

    isThread : *Environment* × *ReferenceType* → *Boolean*

    isThread($\mathcal{JE}$, *ct*) = (*ct*=`Thread`) ∨ (`Thread` ∈ allSuperClasses($\mathcal{JE}$,*ct*))

45. The function length returns the length of a given list:

    length : (τ) -list → *Nat*

    length([ ]) = 0

    length(*v*::*l*) = 1 + length (*l*), ∀ (*v*, *l*) ∈ τ× (τ) -list

46. The function lookupF returns the first field with a given signature found in the superclass hierarchy of a given class in a given environment. If the field is not found the value `None` is returned. Notice that the symbol ⊕ stands for the disjoint union of domains:

lookupF: *Environment*×*FieldSignature*×*ReferenceType* → *Field* ⊕ *NoneType*

lookupF($\mathcal{JE}$,*fs*,*c*) = *f* if retrieveF(*fs*,$\mathcal{JE}$(c).*fields*)=*f* ∧ *f* ≠ "None"

lookupF($\mathcal{JE}$,*fs*,*c*) = lookupF($\mathcal{JE}$,*fs*,$\mathcal{JE}$(*c*).*superClass*)

$$\text{if} \begin{cases} \text{c} \neq \text{"Object"} \\ \text{retrieveF(fs,}\mathcal{JE}\text{(c).}\textit{fields}\text{)} = \text{"None"} \end{cases}$$

lookupF($\mathcal{JE}$,*fs*,*c*) = "None"

$$\text{if} \begin{cases} \text{c} = \text{"Object"} \\ \text{retrieveF(fs,}\mathcal{JE}\text{(c).}\textit{fields}\text{)} = \text{"None"} \end{cases}$$

47. The function lookupM returns the first method with a given signature found in the superclass hierarchy of a given class in a given environment. If the method is not found the value None is returned:

lookupM : *Environment*×*MethodSignature*×*ReferenceType* → *Method* ⊕ *NoneType*

lookupM($\mathcal{JE}$,*ms*,*c*) = *m* if retrieveM(*ms*,$\mathcal{JE}$(c).*methods*)=*m* ∧ *m* ≠ "None"

lookupM($\mathcal{JE}$,*ms*,*c*) = lookupM(*ms*,$\mathcal{JE}$(*c*).*superClass*)

$$\text{if} \begin{cases} \text{c} \neq \text{"Object"} \\ \text{retrieveM(ms,}\mathcal{JE}\text{(c).}\textit{methods}\text{)} = \text{"None"} \end{cases}$$

lookupM($\mathcal{JE}$,*ms*,*c*) = "None"

$$\text{if} \begin{cases} \text{c} = \text{"Object"} \\ \text{retrieveM(ms,}\mathcal{JE}\text{(c).}\textit{methods}\text{)} = \text{"None"} \end{cases}$$

48. The function methodSignatures returns a set of method signatures of all methods in a given list of methods:

methodSignatures : (*Method*)-list → (*MethodSignature*)-set

methodSignatures([ ]) =

methodSignatures(*ml*) = {head(*ml*).*methodSignature*} ∪ methodSignatures(tail(*ml*)) if *ml* ≠ [ ]

49. The function newFrame returns a new frame constructed from a given method, a given program counter, a given list of local variable values a given operand stack and a given synchronized element:

newFrame:*Method*×*ProgramCounter*×*Locals*×*OperandStack*×*SynchronizedElement*→*Frame*

$$\text{newFrame}(m,pc,l,o,z) = frame \text{ if } \begin{cases} frame.method = m \\ frame.programCounter = pc \\ frame.locals = l \\ frame.operandStack = o \\ frame.synchronizedElement = z \end{cases}$$

50. The function newObject returns a new object of a given class in a given environment:

$$\text{newObject} : Environement \times ReferenceType \rightarrow JavaObject$$

$$\text{newObject}(\mathcal{JE},ct) = o \text{ if } \begin{cases} o.classType = ct \\ o.fieldsMap = \text{defaultFieldMap}(\text{allInstanceFields}(\mathcal{JE},ct)) \\ o.monitor = (\text{“None”}, 0, [\,]) \\ o.fromRunnable = \text{“None”} \end{cases}$$

51. The function newThreadInformation returns a new thread structure given a thread stack, a list of locked elements and a thread Id:

$$\text{newThreadInformation}: ThreadStack \times (LockedElement)\text{-}list \times ThreadId \rightarrow ThreadInformation$$

$$\text{newThreadInformation}(s,l,id) = t$$

$$\text{if } \begin{cases} t.threadStack = s \\ t.lockedElements = l \\ t.threadId = id \end{cases}$$

52. The function objectMonitorEntered returns the same object than the given location in the given store but containing the information that the object's monitor has been entered by a given thread:

$$\text{objectMonitorEntered} : Store \times Location \times ThreadId \rightarrow JavaObject$$

$$\text{objectMonitorEntered}(S,Loc,id) = S(Loc)[monitor.waitList \leftarrow \text{suppress}(S(Loc).monitor.waitList,\text{id});$$

$$monitor.threadOwner \leftarrow id; monitor.depth \leftarrow S(loc).monitor.depth+1]$$

53. The function objectMonitorExited returns the same object than the given location in the given store but containing the information that the object's monitor has been exited by a given thread:

objectMonitorExited : *Store*×*Location*×*ThreadId* → *JavaObject*

objectMonitorExited(*S*,*Loc*,*id*) = *S*(*Loc*)[*monitor.threadOwner* ← "None"/ *S*(*Loc*).*monitor.depth*=1;

*monitor.depth* ← *S*(*Loc*).*monitor.depth*-1]

54. The function posStack returns a list obtained from a given list by popping a given number of elements:

posStack : (τ) -list ×*Nat* → (τ) -list

posStack(*l*,0) = *l*

popStack(*l*,*i*) = popStack(tail(*l*),*i*-1)) , ∀ *i* > 0

55. The function pushStack returns a list obtained by appending a given value to a given list:

pushStack : (τ) -list×τ → (τ) -list

pushStack(*l*,*v*) = v::l, ∀ (*v*, *l*) ∈ τ× (τ) -list

56. The function retrieveF searches for a field with a given signature in a given list of fields and returns the field if the field is found otherwise returns the value None:

retrieveF : *FieldSignature*×*Fields* → *Field* ⊕ *NoneType*

retrieveF(*fs*,[ ]) = "None"

retrieveF(*fs*,*l*) = head(*l*) if head(*l*).*fieldSignature* = fs

retrieveF(*fs*,*l*) = retrieveF(*fs*,tail(*l*)) if head(*l*).*fieldSignature* ≠ fs

57. The function retrieveM searches for a method with a given signature in a given list of methods and returns the method if the method has been found otherwise returns the value None:

retrieveM : *MethodSignature*×*Methods* → *Method* ⊕ *NoneType*

retrieveM(*ms*,[ ]) = "None"

retrieveM(*ms*,*l*) = head(*l*) if head(*l*).*methodSignature* = ms

retrieveM(*ms*,*l*) = retrieveM(*ms*,tail(*l*)) if head(*l*).*methodSignature* ≠ ms

58. The function suppress returns a list constructed from a given list by suppressing all the occurrences of a given value :

suppress : $\tau\times$ ($\tau$) -list $\rightarrow$ ($\tau$) -set

suppress(v,[ ]) = [ ]

suppress(v,l) = suppress(v,tail(l)); if head(l)=v

suppress(v,l) = head(l)::suppress(v,tail(l)); if head(l) $\neq$ v

59. The function tail returns the tail of a given list:

tail : ($\tau$) -list $\rightarrow$ ($\tau$) -list

tail([ ]) = [ ]

tail($v$::$l$) = $l$, $\forall$ ($v$, $l$) $\in \tau\times$ ($\tau$) -list

60. The function setOf returns a set of elements of a given list:

setOf : ($\tau$) -list $\rightarrow$ ($\tau$) -set

setOf([ ]) =

setOf(v::l) = {v} $\cup$ setOf(l)

61. The function thisConstantPoolEntry returns a constant pool entry given an environment, a method and an index for the entry:

thisConstantPoolEntry : *Environment*$\times$*Method*$\times$*Nat* $\rightarrow$ *ConstantPoolEntry*

thisConstantPoolEntry($\mathcal{JE}$,$m$,$i$) = $\mathcal{JE}$($m$.*fromClass*).*constantPool*($i$)

62. The function typeOfField returns the type of a field:

typeOfField : *Field* $\rightarrow$ *Type*

typeOfField($f$) = $f$.*fieldSignature.type*

63. The function waitingThreads returns a set of thread Ids waiting for a list of objects or classes described in a given list of locations or class identifiers:

waitingThreads : *Store*$\times$*Environment*$\times$ (*ClassOrLocation*) -list $\rightarrow$ (*ThreadId*) -set

waitingThreads($S$, $\mathcal{JE}$, [ ]) = $\emptyset$

waitingThreads($S$, $\mathcal{JE}$, x::l) = setOf(S(x).*waitList*) $\cup$ waitingThreads($S$, $\mathcal{JE}$, tail(l)) if x $\in$ Dom(S)

waitingThreads($S$, $\mathcal{JE}$, x::l) = setOf($\mathcal{JE}$(x).*monitor.waitList*) $\cup$ waitingThreads($S$, $\mathcal{JE}$, tail(l)) if x $\in$ $\mathcal{JE}$(S)

# Appendix II: AspectJ Semantics Utility Functions

1. The function argSTest takes an environment, a natural number(the current argument position), a first type list (types of the arguments), a second type list (types to match) and returns the type matching test. The function argTest generates the test for one argument whereas argSTest generates the test for a list of arguments. These functions are used in the pointcut matching process when the pointcut is an "args" pointcut:

   argSTest : *Environment*×*Nat*× (*Type*) `-list`× (*Type*) `-list`→*Test*

   argSTest($\mathcal{E}$,*i*,[ ],[ ])=`always`

   argSTest($\mathcal{E}$,*i*,$l_1$,$l_2$)=`never` if length($l_1$) $\neq$ length($l_2$)

   argSTest($\mathcal{E}$,i,$l_1$,$l_2$) = makeAnd(argTest($\mathcal{E}$,$i$,head($l_1$),head($l_2$)),argSTest($\mathcal{E}$,$i+1$,tail($l_1$),tail($l_2$)))

   if length($l_1$) = length($l_2$)

   argTest : *Environment*×*Nat*×*Type*×*Type*→*Test*

   argTest($\mathcal{E}$,*i*,$t_1$,$t_2$)=`never` if $t_1 \neq t_2$ and $t_2 \notin$ allSuperClasses($\mathcal{E}$,$\{t_1\}$)

   argTest($\mathcal{E}$,*i*,$t_1$,$t_2$)=`always` if $t_1 = t_2$

   argTest($\mathcal{E}$,*i*,$t_1$,$t_2$)=`arg(`*i*`) instanceof (`$t_2$`)` if $t_1 \in$ allSuperClasses($\mathcal{E}$,$\{t_2\}$)

2. The function argumentTypes takes a shadow as argument and returns the list of the types of its arguments:

   argumentsTypes : *Shadow*→ (*Type*) `-list`

   argumentTypes(*s*)=*s.signature.argumentsType* if *s.kind* $\neq$ `field_get` $\wedge$ *s.kind* $\neq$ `field_set`

   argumentTypes(*s*)=[*s.signature.type*] if *s.kind* = `field_get` $\vee$ *s.kind* = `field_set`

3. The function branchIns takes a JVML instruction and returns a boolean that indicates if the JVML instruction is a branching instruction or not:

branchIns : *JVMLInst*→*Boolean*

branchIns(*ins*)= true if *ins*=`goto` *adr* ∨ *ins*=`ifeq` *adr* ∨ *ins*=`ifne` *adr*

branchIns(*ins*)= false otherwise

4. The function changeMethods changes a given method contained in a given list by another given method:

   changeMethods: (*Method*) -list×*Method*×*Method* → (*Method*) -list

   changeMethods($l$,*m*,*m*′) =head($l$)::changeMethods(tail($l$),*m*,*m*′) if head($l$).*signature* ≠ *m.signature*

   changeMethods($l$,*m*,*m*′) =*m*′::tail($l$) if head($l$).*signature*=*m.signature*

5. The function enclosingShadow returns the first Shadow of a given method:

   enclosingShadow : *Method* → `Shadow`

   enclosingShadow(*m*) = ((`construct_execut`,`false`),*m.methodSignature*,*m.modifiers*,*m.fromClass*,0,
   *m*.lastNumInst(*m.code*)+2),[ ]) if *m.methodSignature.name*="`init`"

   enclosingShadow(*m*) = ((`static_init`,`false`),*m.methodSignature*,*m.modifiers*,*m.fromClass*,0,
   *m*.lastNumInst(*m.code*)+2),[ ]) if *m.methodSignature.name*="`clinit`"

   enclosingShadow(*m*) = ((`advice_execut`,`false`),*m.methodSignature*,*m.modifiers*,*m.fromClass*,0,
   *m*.lastNumInst(*m.code*)+2),[ ]) if *m.fromClass.aspect*=1

   enclosingShadow(*m*) = ((`method_execut`,`false`),*m.methodSignature*,*m.modifiers*,*m.fromClass*,0,
   *m*.lastNumInst(*m.code*)+2),[ ]) otherwise.

6. The function freeInstructions takes a code and two natural numbers $i$ and $pc$ and returns a new code. The new code is exactly identical to the given one except that the instructions are numbered differently. We add the value $i$ to all the instruction numbers that are greater than $pc$:

   freeInstructions : *Code*×*Nat*×*Nat* →*Code*

   freeInstructions($c$,$i$,$pc$)=$c'$ if

$$\begin{cases} c'(k) = c(k)\ \forall k \in Dom(c)\ /\ k <= pc \\ c'(k+i) = c(k)\ \forall k \in Dom(c)\ /\ k > pc \end{cases}$$

7. The function genPool takes an environment, a method and a component type. If the class represented by the component type is found in the class constant pool of the given method, the function will return the respective entry number and the given environment without changing it. If the class represented by the component type is not found in the class constant pool of the given method, genPool generates a new entry in the class constant pool of the given method and returns the new constant pool entry number with the environment changed:

   genPool:$Environment \times Method \times ComponentType \rightarrow Nat \times Environment$

   genPool($\mathcal{E}$,$m$,$ct$)=($k$,$\mathcal{E}$) if $\mathcal{E}.javaEnvironment(m.fromClass).constantPool(k)=ct$

   where $k \in$ Dom($\mathcal{E}.javaEnvironment(m.fromClass).constantPool$)

   genPool ($\mathcal{E}$,$m$,$ct$)=($k$,$\mathcal{E}$[$javaEnvironment \leftarrow \mathcal{E}.javaEnvironment[m.fromClass \mapsto$

   $\mathcal{E}.javaEnvironment(m.fromClass)[constantPool \leftarrow \mathcal{E}.javaEnvironment(m.fromClass).constantPool.[k \mapsto ct]]]]$)

   where $k \notin$ Dom($\mathcal{E}.javaEnvironment(m.fromClass).constantPool$)

8. . The function getArgVar takes a method, a shadow, two natural numbers as arguments and returns a position in the method local variable. The first given number corresponds to the maximun length of the method local variable and the second number indicates the argument. Hence getArgVar($m$,$s$,$maxLocals$,$i$) will return the emplacement where the argument number $i$ of the shadow $s$ has been stored in the method local variable table of $m$:

   getArgVar : $Method \times Shadow \times Nat \times Nat \rightarrow Nat$

   getArgVar($m$,$s$,$maxLocals$,$i$)=$maxLocals$-($i$-1) if $\neg$ hasTarget($s$,$m$)

   getArgVar($m$,$s$,$maxLocals$,$i$)=($maxLocals$-1)-($i$-1) if hasTarget($s$,$m$)

9. The function hasTarget takes a shadow and a method and returns true if the given shadow has a "target" object:

   hasTarget: $Shadow \times Method \rightarrow Boolean$

   hasTarget($s$,$m$)=( $\neg$ neverHasTarget(s) $\wedge$ hasThis(s,m)) if isTargetSameAsThis(s)

   hasTarget($s$,$m$)=( $\neg$ neverHasTarget(s) $\wedge$ `static` $\notin$ s.$modifiers$) if $\neg$ (isTargetSameAsThis(s))

10. The function hasThis takes a shadow and a method and returns true if the shadow in this method

has a "this" object:

hasThis: $Shadow \times Method \rightarrow Boolean$

hasThis($s$,$m$) =( $\neg$ neverHasThis($s$) $\wedge$ `static` $\notin m.modifiers$)

11. The function neverHasThis takes a shadow and returns true if the shadow has never a "this" object:

    neverHasThis: $Shadow \rightarrow Boolean$

    neverHasThis($s$) =($\pi_1$($s.kind$)=`static_init`)

12. The function insertAdvice takes an environment, a method, and a list of shadows and returns a new environment and a new method (where the advice has been injected):

    insertAdvice : $Environment \times Method \times$ ($Shadow$) -list $\rightarrow Environment \times Method$

    insertAdvice($\mathcal{E}$,$m$,$msh$)=insertAfterAdvice($\mathcal{E}$,$m$,$msh$)

    if head(head($msh$).$mungers$).$adviceInfo.akind$=`After`

    insertAdvice($\mathcal{E}$,$m$,$msh$)=insertBeforeAdvice($\mathcal{E}$,$m$,$msh$)

    if head(head($msh$).$mungers$).$adviceInfo.akind$=`Before`

13. The function insertAfterAdvice takes an environment, a method, and a list of shadows and returns a new environment and a new method (where the advice has been injected). It injects the advice in the method by injecting two JVML instructions: The call to the static "`aspectOf`" method of the advice aspect, and the advice call itself. This corresponds to the injection of two bytecodes: `invokestatic` $i$ and `invokevirtual` $j$ where $i$ and $j$ are added as new entries to the constant pool of the method class. It is necessary to call the static "`aspectOf`" method of the aspect to obtain an instance for use as the receiver of the advice call. The "`aspectOf`" method is automatically generated when compiling the aspect into a class:

insertAfterAdvice : $Environment \times Method \times (Shadow)\text{-}list \rightarrow Environment \times Method$

insertAfterAdvice($\mathcal{E}$,$m$,$msh$)=($\mathcal{E}$',$m$') if

$$
\left\{
\begin{array}{ll}
 & \pi_2(\mathsf{head}(\mathrm{msh}).kind) = \mathtt{true}\ //not\ execution\ shadow \\
\wedge & \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge & c = \mathcal{JE}(m.fromClass) \\
\wedge & cpool = c.constantPool\ //Getting\ the\ Constant\ pool\ of\ the\ class\ of\ m. \\
\wedge & ad = \mathsf{head}(\mathsf{head}(\mathrm{msh}).\mathrm{mungers}).adviceInfo \\
\wedge & ms = \mathsf{signatureAspectOf}(ad) \\
\wedge & cpool1 = cpool[i \mapsto \mathsf{newPoolEntry}(ms, ad.fromClass)],\ i \notin Dom(cpool) \\
\wedge & cpool2 = cpool1[j \mapsto \mathsf{newPoolEntry}(ad.adviceSignature, ad.fromClass)]\, j \notin Dom(cpool1) \\
\wedge & pc = \mathsf{head}(\mathrm{msh}).end \\
\wedge & code1 = \mathsf{mergeInstructions}(m.code, [\mathtt{invokestatic}\ i, \mathtt{invokevirtual}\ j], pc+1) \\
\wedge & m' = m[\mathrm{code} \leftarrow code1]\ //Setting\ the\ code\ of\ m\ with\ the\ updates. \\
\wedge & c1 = c[constantPool \leftarrow cpool2, methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\
\wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1]\ //Updating\ the\ class\ where\ m\ is\ defined. \\
\wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{array}
\right.
$$

insertAfterAdvice($\mathcal{E}$,$m$,$msh$) =($\mathcal{E}$',$m$') if

$$
\begin{cases}
\pi_2(\text{head}(\text{msh}).kind) = \texttt{false} \;//\; execution\ shadow \\
\wedge\ \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge\ c = \mathcal{JE}(m.fromClass) \\
\wedge\ cpool = c.constantPool \\
\wedge\ ms = \text{signatureAspectOf}(ad) \\
\wedge\ cpool1 = cpool[i \mapsto \text{newPoolEntry}(ms, ad.fromClass)]\ i \notin Dom(cpool) \\
\wedge\ ad = \text{head}(\text{head}(\text{msh}).mungers).adviceInfo \\
\wedge\ cpool2 = cpool1[j \mapsto \text{newPoolEntry}(ad.adviceSignature, ad.fromClass)]\ j \notin Dom(cpool1) \\
\wedge\ pc = \text{head}(\text{msh}).end \\
\wedge\ code1 = \text{mergeInstructions}(m.code, [\texttt{invokestatic}\ i, \texttt{invokevirtual}\ j], pc-3) \\
\wedge\ m' = m[code \leftarrow code1] \\
\wedge\ c1 = c[constantPool \leftarrow cpool2, methods \leftarrow \text{changeMethods}(c1.methods, m, m')] \\
\wedge\ \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \\
\wedge\ \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{cases}
$$

14. The function insertAfterStore takes an environment, a method, a list of shadows, and a list of JVML instructions and returns a new environment and a new method (where the JVML instructions have been injected). It injects the given instructions in the method, updates the environment and readjusts the values of the *start* and *end* in the shadows:

insertAfterStore : *Environment*×*Method*× (*Shadow*) -*list*× (*Instruction*) -`list`

$\rightarrow$*Environment*×*Method*

insertAfterStore($\mathcal{E}$,*m*,*msh*,*il*)=($\mathcal{E}$',*m*') if

$$\left\{\begin{array}{ll} & \pi_2(\mathsf{head}(\mathrm{msh}).kind) = \texttt{true}\ //\textit{not execution shadow} \\ \wedge & \mathcal{JE} = \mathcal{E}.javaEnvironment \\ \wedge & c = \mathcal{JE}(m.fromClass) \\ \wedge & pc = \mathsf{head}(\mathrm{msh}).end \\ \wedge & code1 = \mathsf{mergeInstructions}(m.code, il, pc+1) \\ \wedge & m' = m[\mathrm{code} \leftarrow code1]\ //\textit{Setting the code of m with the updates.} \\ \wedge & c1 = c[methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\ \wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1]\ //\textit{Updating the class where m is defined.} \\ \wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1] \end{array}\right.$$

insertAfterStore($\mathcal{E}$,*m*,*msh*) =($\mathcal{E}$',*m*',*msh*') if

$$\left\{\begin{array}{ll} & \pi_2(\mathsf{head}(\mathrm{msh}).kind) = \texttt{false}\ //\ \textit{execution shadow} \\ \wedge & \mathcal{JE} = \mathcal{E}.javaEnvironment \\ \wedge & c = \mathcal{JE}(m.fromClass) \\ \wedge & pc = \mathsf{head}(msh).end \\ \wedge & code1 = \mathsf{mergeInstructions}(m.code, il, pc-1-\mathsf{length}(il)) \\ \wedge & m' = m[\mathrm{code} \leftarrow code2] \\ \wedge & c1 = c[constantPool \leftarrow cpool2, methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\ \wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \\ \wedge & msh' = \mathsf{changeShadows}(msh, pc, 2) \\ \wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1] \end{array}\right.$$

15. The function insertBeforeAdvice takes an environment, a method, a program counter, and an advice as arguments and returns a new environment and a new method. It works as the insertAfterAdvice except that the injection is done before the shadow:

$$\begin{array}{l}
\textsf{insertBeforeAdvice} : \mathit{Environment} \times \mathit{Method} \times (\mathit{Shadow})\text{-}\mathit{list} \rightarrow \mathit{Environment} \times \mathit{Method} \\
\textsf{insertBeforeAdvice}(\mathcal{E}, m, msh) = (\mathcal{E}', m') \text{ if} \\
\left\{ \begin{array}{ll}
 & \mathcal{JE} = \mathcal{E}.\mathit{javaEnvironment} \\
\wedge & c = \mathcal{JE}(m.\mathit{fromClass}) \\
\wedge & \mathit{cpool} = c.\mathit{constantPool} \\
\wedge & ms = \textsf{signatureAspectOf}(ad) \\
\wedge & \mathit{cpool1} = \mathit{cpool}[i \mapsto \textsf{newPoolEntry}(ms, ad.\mathit{fromClass})]\ i \notin \mathit{Dom}(\mathit{cpool}) \\
\wedge & ad = \textsf{head}(\textsf{head}(\text{msh}).\mathit{mungers}).\mathit{adviceInfo} \\
\wedge & \mathit{cpool2} = \mathit{cpool1}[j \mapsto \textsf{newPoolEntry}(ad.\mathit{adviceSignature}, ad.\mathit{fromClass})] j \notin \mathit{Dom}(\mathit{cpool1}) \\
\wedge & pc = \textsf{head}(\text{msh}).\mathit{start} \\
\wedge & \mathit{code1} = \textsf{mergeInstructions}(m.\mathit{code}, [\texttt{invokestatic}\ i, \texttt{invokevirtual}\ j], pc+1) \\
\wedge & m' = m[\mathit{code} \leftarrow \mathit{code1}] \\
\wedge & c1 = c[\mathit{constantPool} \leftarrow \mathit{cpool2}, \mathit{methods} \leftarrow \textsf{changeMethods}(c1.\mathit{methods}, m, m')] \\
\wedge & \mathcal{JE}1 = \mathcal{JE}[m.\mathit{fromClass} \mapsto c1] \\
\wedge & \mathcal{E}' = \mathcal{E}[\mathit{javaEnvironment} \leftarrow \mathcal{JE}1]
\end{array} \right.
\end{array}$$

16. The function insertImpdep1 takes an environment, a method and a program counter and returns the environment and the method but with inserting two instructions `impdep1` before the instruction at the given program counter:

insertImpdep1 : $Environment \times Method \times ProgramCounter \rightarrow Environment \times Method$

insertImpdep1($\mathcal{E}$,$m$,$pc$) =($\mathcal{E}$',$m$') if

$$
\left\{
\begin{array}{ll}
 & m'(i) = m(i), \forall i \in Dom(m) / 0 <= i < pc \\
\wedge & m'(pc) = \texttt{impdep1} \\
\wedge & m'(pc+1) = m(pc) \\
\wedge & m'(pc+2) = \texttt{impdep1} \\
\wedge & m'(i+2) = m(i), \forall i \in Dom(m) / i >= pc+1 \\
\wedge & \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge & c = \mathcal{JE}(m.fromClass) \\
\wedge & c1 = c[methods \leftarrow \mathsf{changeMethods}(c.methods, m, m')] \\
\wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \\
\wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{array}
\right.
$$

17. The function insertStore takes an environment, a method, a list of shadows, and a list of JVML instructions and returns a new environment and a new method (where the JVML instructions have been injected). It injects the given instructions in the method, updates the environment and readjusts the values of the *start* and *end* in the shadows:

insertStore : *Environment*×*Method*× (*Shadow*) -*list*× (*Instruction*) -list

→*Environment*×*Method*

insertStore($\mathcal{E}$,*m*,*msh*,*il*)=($\mathcal{E}$',*m*') if

$$\begin{cases} \mathcal{JE} = \mathcal{E}.javaEnvironment \\ \wedge\ c = \mathcal{JE}(m.fromClass) \\ \wedge\ pc = \mathsf{head}(\mathrm{msh}).start \\ \wedge\ code1 = \mathsf{mergeInstructions}(m.code, il, pc{+}1) \\ \wedge\ m' = m[\mathrm{code} \leftarrow code1]\ //Setting\ the\ code\ of\ m\ with\ the\ updates. \\ \wedge\ c1 = c[methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\ \wedge\ \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1]\ //Updating\ the\ class\ where\ m\ is\ defined. \\ \wedge\ \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1] \end{cases}$$

18. The function insertTestAfterInstructions takes an environment, a method, and a list of shadows, and returns a new environment and a new method (where the JVML instructions for the dynamic test have been injected):

insertTestAfterInstructions : $Environment \times Method \times (Shadow)\text{-}list \times Nat \rightarrow Environment \times Method$

insertTestAfterInstructions($\mathcal{E}$,$m$,$msh$,$maxLocals$)=($\mathcal{E}$',$m$') if

$$
\left\{
\begin{array}{ll}
 & \pi_2(\mathsf{head}(msh).kind) = \texttt{true} \ //\textit{not execution shadow} \\
\wedge & \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge & c = \mathcal{JE}(m.fromClass) \\
\wedge & s = \mathsf{head}(msh) \\
\wedge & test = \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest \\
\wedge & l = \mathsf{lenTestCode}(test) \\
\wedge & yes = \mathsf{head}(msh).end + l + 1 \\
\wedge & no = \mathsf{head}(msh).end + l + 3 \\
\wedge & start = \mathsf{head}(msh).end + 1 \\
\wedge & code_1 = \mathsf{getTestInstructions}(\mathcal{E}, m, s, test, yes, no, start, maxLocals) \\
\wedge & code_2 = m.\mathsf{code} \dagger code_1 \\
\wedge & m' = m[\mathsf{code} \leftarrow code_2] \ //\textit{Setting the code of } m \textit{ with the updates.} \\
\wedge & c1 = c[methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\
\wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \ //\textit{Updating the class where } m \textit{ is defined.} \\
\wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{array}
\right.
$$

insertTestAfterInstructions($\mathcal{E}$,$m$,msh,$maxLocals$)=($\mathcal{E}$',$m$') if

$$
\begin{cases}
\pi_2(\mathsf{head}(msh).kind) = \texttt{false} \;//\; execution\ shadow \\
\mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge\ c = \mathcal{JE}(m.fromClass) \\
\wedge\ s = \mathsf{head}(msh) \\
\wedge\ test = \mathsf{head}(\mathsf{head}(msh).mungers).pointcutTest \\
\wedge\ l = \mathsf{lenTestCode}(test) \\
\wedge\ yes = \mathsf{head}(msh).end - 3 \\
\wedge\ no = \mathsf{head}(msh).end - 1 \\
\wedge\ start = \mathsf{head}(msh).end - 2 - l \\
\wedge\ code_1 = \mathsf{getTestInstructions}(\mathcal{E}, m, s, test, yes, no, start, maxLocals) \\
\wedge\ code_2 = m.code \dagger code_1 \\
\wedge\ m' = m[\mathsf{code} \leftarrow code_2]\ //Setting\ the\ code\ of\ m\ with\ the\ updates. \\
\wedge\ c1 = c[methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\
\wedge\ \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1]\ //Updating\ the\ class\ where\ m\ is\ defined. \\
\wedge\ \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{cases}
$$

19. The function insertTestBeforeInstructions takes an environment, a method, and a list of shadows, and returns a new environment and a new method (where the JVML instructions for the dynamic test have been injected):

insertTestBeforeInstructions : $Environment \times Method \times (Shadow)\text{-}list \times Nat \rightarrow$

$Environment \times Method$

insertTestBeforeInstructions($\mathcal{E}$,$m$,$msh$,$maxLocals$)=($\mathcal{E}$',$m$') if

$$
\left\{
\begin{array}{ll}
 & \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge & c = \mathcal{JE}(m.fromClass) \\
\wedge & s = \mathsf{head}(msh) \\
\wedge & test = \mathsf{head}(\mathsf{head}(msh).\mathrm{mungers}).\mathrm{pointcutTest} \\
\wedge & l = \mathsf{lenTestCode}(test) \\
\wedge & yes = \mathsf{head}(msh).start + l \\
\wedge & no = \mathsf{head}(msh).start + l + 2 \\
\wedge & start = \mathsf{head}(msh).start + 1 \\
\wedge & code_1 = \mathsf{getTestInstructions}(\mathcal{E}, m, s, test, yes, no, start, maxLocals) \\
\wedge & code_2 = m.\mathrm{code} \dagger code_1 \\
\wedge & m' = m[\mathrm{code} \leftarrow code_2] \ //Setting\ the\ code\ of\ m\ with\ the\ updates. \\
\wedge & c1 = c[methods \leftarrow \mathsf{changeMethods}(c1.methods, m, m')] \\
\wedge & \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \ //Updating\ the\ class\ where\ m\ is\ defined. \\
\wedge & \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1]
\end{array}
\right.
$$

20. The function isInstShadow returns true if the given instruction corresponds to a shadow otherwise it returns false:

    isInstShadow :$Environment \times Method \times Instruction \rightarrow Boolean$

    isInstShadow($\mathcal{E}$,$m$,ins) =(kindOfShadow($\mathcal{E}$,$m$,ins)$\neq$None)

21. The function isTargetSameAsThis takes a shadow and returns true if the "this" object is the same that the "target" object for this shadow:

isTargetSameAsThis: *Shadow* → *Boolean*

isTargetSameAsThis($s$)=( ($\pi_1(s.kind)$=method_execut)

$\vee$( $\pi_1(s.kind)$=construct_execut)

$\vee$( $\pi_1(s.kind)$=static_init)

$\vee$( $\pi_1(s.kind)$=advice_execut) )

22. The function kindOfShadow returns the kind of the instruction shadow in case where the instruction is a shadow otherwise it returns None:

kindOfShadow : *Instruction*×*Signature* → *Kind* ⊕ *NoneType*

kindOfShadow($ins$,$s$) =(method_call,true)

$$\text{if} \begin{cases} ins = \texttt{invokevirtual}\ i \\ \vee(ins = \texttt{invokespecial}\ i \wedge s.name \neq \texttt{init} \\ \vee ins = \texttt{invokestatic}\ i \\ \vee ins = \texttt{invokeinterface}\ i,n \end{cases}$$

kindOfShadow($ins$,$s$) =(field_get,true)

$$\text{if} \begin{cases} ins = \texttt{getfield}\ i \\ \vee ins = \texttt{getstatic}\ i \end{cases}$$

kindOfShadow($ins$,$s$) =(field_set,true)

$$\text{if} \begin{cases} ins = \texttt{putfield}\ i \\ \vee ins = \texttt{putstatic}\ i \end{cases}$$

kindOfShadow($ins$,$s$) =(construct_call,true)

$$\text{if} \begin{cases} ins = \texttt{invokespecial}\ i \\ \wedge s.name = \texttt{init} \end{cases}$$

kindOfShadow($ins$,$s$) =None otherwise.

23. The function lastNumInst returns the program counter of the last instruction in a given code:

lastNumInst : *Code* → Nat

lastNumInst($c$) =maximum(Dom($c$))

24. The function lenTestCode takes a test and returns the number of instructions relative to this test. Notice that for each basic dynamic test, four instructions are needed: {`aload`, `instanceof`, `ifeq`, `goto`}:

lenTestCode : *Test* → *Nat*

lenTestCode(`And`($t_1$,$t_2$))=lenTestCode($t_1$)+lenTestCode($t_2$)

lenTestCode(`Or`($t_1$,$t_2$))=lenTestCode($t_1$)+lenTestCode($t_2$)

lenTestCode(`Not`($t_1$)=lenTestCode($t_1$))

lenTestCode($t$)=4 if

$$\begin{cases} t = \texttt{this instanceof } (ct) \\ \vee t = \texttt{target instanceof}(ct) \\ \vee t = \texttt{args}(i) \texttt{ instanceof}(ct) \end{cases}$$

25. The function liberate takes an environment, a method, a list of shadows, and a number indicating the number of instructions that we want to let free after or before the current shadow of the method:

liberate : *Environment*×*Method*× (*Shadow*) *-list*×*Nat*

→*Environment*×*Method*× (*Shadow*) *-list*

liberate($\mathcal{E}$,*m*,*msh*,*i*)=liberateAt($\mathcal{E}$,*m*,*msh*,*i*,head(msh).*start*) if

$$\begin{cases} \text{head(head}(msh).mungers).adviceInfo.akind = \texttt{Before} \end{cases}$$

liberate($\mathcal{E}$,*m*,*msh*,*i*)=liberateAt($\mathcal{E}$,*m*,*msh*,*i*,head(msh).*end*) if

$$\begin{cases} \text{head(head}(msh).mungers).adviceInfo.akind = \texttt{After} \\ \wedge \pi 2(\text{head}(msh).kind) = \texttt{false}//\textit{not execution shadow} \end{cases}$$

liberate($\mathcal{E}$,*m*,*msh*,*i*)=liberateAt($\mathcal{E}$,*m*,*msh*,*i*,head(msh).*end*-2) if

$$\begin{cases} \text{head(head}(msh).mungers).adviceInfo.akind = \texttt{After} \\ \wedge \pi 2(\text{head}(msh).kind) = \texttt{true}//\textit{execution shadow} \end{cases}$$

26. The function liberateAt takes an environment, a method, a list of shadows, a number indicating the number of instructions that we want to let free, and another number that indicates the position from

which we want to free the instructions:

liberateAt : *Environment*×*Method*× (*Shadow*) -*list*×*Nat*×*Nat*

→*Environment*×*Method*× (*Shadow*) -*list*

liberateAt($\mathcal{E}$,*m*,*msh*,*i*,*pc*)=($\mathcal{E}$',*m*',*msh*') if

$$\begin{cases} \mathcal{JE} = \mathcal{E}.javaEnvironment \\ \wedge c = \mathcal{JE}(m.fromClass) \\ \wedge code1 = \text{translate}(\text{freeInstructions}(m.code, i, pc), i, pc) \\ \wedge m' = m[code \leftarrow code1] \ //Setting\ the\ code\ of\ m\ with\ the\ updates. \\ \wedge c1 = c[methods \leftarrow \text{changeMethods}(c1.methods, m, m')] \\ \wedge \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \ //Updating\ the\ class\ where\ m\ is\ defined. \\ \wedge \mathcal{E}' = \mathcal{E}[javaEnvironment \leftarrow \mathcal{JE}1] \\ \wedge msh' = \text{changeShadows}(msh, pc, i) \end{cases}$$

27. The following function loadCode takes a list of store instructions and returns a list of load instructions that allow to reload the stored variables onto the stack:

loadCode: (*Instruction*) -list→ (*Instruction*) -list

loadCode(*l*::astore_*i*)=aload_*i*::loadCode(*l*)

loadCode(*l*::istore_*i*)=iload_*i*::loadCode(*l*)

28. The function match takes an environment, a method, a shadow, a list of advices and returns the same shadow except that the part *mungers* of the shadow reflects all the shadow mungers that match with it:

match : *Environment*×*Method*×*Shadow*× (*AdviceInfo*) -*list*→*Shadow*

match($\mathcal{E}$,*m*,*s*,*l*)=*s'* if

$$\begin{cases} s'.kind = s.kind \\ \wedge\ s'.signature = s.signature \\ \wedge\ s'.start = s.start \\ \wedge\ s'.end = s.end \\ \wedge\ s'.mungers = l'/l' = \text{matched}(\mathcal{E},m,s,l) \end{cases}$$

29. The function matched takes an environment, a method, a shadow, a list of advices and returns a list of shadow mungers that matches with the shadow:

matched : *Environment*×*Method*×*Shadow*× (*AdviceInfo*) -*list*→ (*ShadowMunger*) -*list*

matched($\mathcal{E}$,*m*,*s*,[ ])=[ ]

matched($\mathcal{E}$,*m*,*s*,*a*::*l*)=matched($\mathcal{E}$,*m*,*s*,*l*)

if matchPcut($\mathcal{E}$,*m*,*s*,*a.pointcut*)=never

matched($\mathcal{E}$,*m*,*s*,*a*::*l*)=*a'*::matched($\mathcal{E}$,*m*,*s*,*l*)

$$\text{if} \begin{cases} \text{matchPcut}(\mathcal{E},m,s,a.pointcut) \neq \texttt{never} \\ \wedge\ a'.adviceInfo = a \\ \wedge\ a'.pointcutTest = \text{matchPcut}(\mathcal{E},m,s,a.pointcut) \end{cases}$$

30. The function mergeInstructions takes a code, a list of JVML instructions, and a natural number representing a position and returns a new code. The new code is the result of inserting the list of JVML instructions to the given code at the given position:

mergeInstructions : $Code \times (Instruction)\text{-}list \times Nat \rightarrow Code$

mergeInstructions($c$,[ ],$n$)=$c$

mergeInstructions($c$,$i$::$il$,$n$)=mergeInstructions(mergeOneInstruction($c$,$i$,$n$),$il$,$n$+1)

31. The function mergeOneInstruction takes a code, a JVML instruction, and a natural number representing a position and returns a new code. The new code is the result of inserting the JVML instruction to the given code at the given position:

mergeOneInstruction : $Code \times Instruction \times Nat \rightarrow Code$

mergeOneInstruction($c$,$i$,$n$)=$c'$ if

$$\begin{cases} c'(k) = c(k)\ \forall k \in Dom(c)\ /\ k < n \\ c'(n) = i \\ c'(k) = c(k)\ \forall k \in Dom(c)\ /\ k > n \end{cases}$$

32. The function neverHasThis takes a shadow and returns true if the shadow has never a "this" object:

neverHasThis: $Shadow \rightarrow Boolean$

neverHasThis($s$) =($\pi_1$($s.kind$)=`static_init`)

33. The function newPoolEntry returns a constant pool entry for a method given the signature of the method and its class:

newPoolEntry : $MethodSignature \times Class \rightarrow ConstantPoolEntry$

newPoolEntry($ms$,$ct$) = $c$ if

$$\begin{cases} c.methodSignature = ms \\ \wedge c.supposedClass = ct \end{cases}$$

34. The function newShadow returns a shadow given an environment, a method, a shadow instruction, starting and ending positions :

newShadow : *Environment*×*Method*×*ShadInst*×nat×nat×

(*ShadowMunger*) -list→*Shadow*

newShadow($\mathcal{E}$,*m*,*ins*,*b*,*e*,*l*)=*s* if

$$
\begin{cases}
(ins = \texttt{invokevirtual}\ i \vee ins = \texttt{invokespecial}\ i \vee \\
ins = \texttt{invokestatic}\ i \vee ins = \texttt{invokeinterface}\ i\ n) \\
\wedge \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge c = \mathcal{JE}(m.fromClass) \\
\wedge cpool = c.constantPool \\
\wedge cpoolentry = cpool(i) \\
\wedge m = \text{retrieveM}(cpoolentry.supposedClass.methods, cpoolentry.methodSignature) \\
\wedge s.signature = m.signature \\
\wedge s.kind = \text{kindOfShadow}(ins, s.signature) \\
\wedge s.modifiers = m.modifiers \\
\wedge s.fromClass = m.fromClass \\
\wedge s.start = b \\
\wedge s.end = e \\
\wedge s.mungers = l
\end{cases}
$$

newShadow($\mathcal{E}$,*m*,*ins*,*b*,*e*,*l*)=*s* if

$$\left\{\begin{array}{l}
(ins = \texttt{getfield}\ i \vee ins = \texttt{putfiled}\ i \vee ins = \texttt{getstatic}\ i \vee ins = \texttt{putstatic}\ i) \\
\wedge \mathcal{JE} = \mathcal{E}.javaEnvironment \\
\wedge c = \mathcal{JE}(m.fromClass) \\
\wedge cpool = c.constantPool \\
\wedge cpoolentry = cpool(i) \\
\wedge f = \text{retrieveF}(cpoolentry.supposedClass.fields, cpoolentry.fieldSignature) \\
\wedge s.signature = f.signature \\
\wedge s.kind = \text{kindOfShadow}(ins, s.signature) \\
\wedge s.modifiers = f.modifiers \\
\wedge s.fromClass = f.fromClass \\
\wedge s.start = b \\
\wedge s.end = e \\
\wedge s.mungers = l
\end{array}\right.$$

35. The function oneIsDynamic returns true if one of the given mungers has a pointcut test different of `always`:

    oneIsDynamic : (*ShadowMunger*) `-list` → *Boolean*

    oneIsDynamic([ ])= `false`

    oneIsDynamic(*l*)= `true` if head(*l*).*pointcutTest* ≠ `always`

    oneIsDynamic(*l*)= oneIsDynamic(tail(*l*)) if `head(`*l*`).`*pointcutTest* = `always`

36. The function preShadowing takes an environment and a method and returns the given environment and method but with the execution shadow wrapped by `impdep1` mnemonics and a list containing this shadow:

preShadowing : $Environment \times Method \rightarrow Environment \times Method \times (Shadow)$ *-list*

preShadowing($\mathcal{E}$,$m$) =($\mathcal{E}'$,$m'$,l) if

$$\left\{\begin{array}{l} k = \text{lastNumInst(m.}code\text{)} \\ \wedge m'.code(i+1) = m.code(i), \forall i \in Dom(m.code) \\ \wedge m'.code(0) = \texttt{impdep1} \\ \wedge m'.code(k+2) = \texttt{impdep1} \\ \wedge l = [\text{enclosingShadow}(m)] \\ \wedge \mathcal{JE} = \mathcal{E}.javaEnvironment \\ \wedge c = \mathcal{JE}(m.fromClass) \\ \wedge c1 = c[methods \leftarrow \text{ChangeMethods}(c.methods, m, m')] \\ \wedge \mathcal{JE}1 = \mathcal{JE}[m.fromClass \mapsto c1] \\ \wedge \mathcal{E}' = \mathcal{E}[\text{javaEnvironment} \leftarrow \mathcal{JE}1] \end{array}\right.$$

37. The function shadowing takes an environment, a method, a program counter and a list of shadows and returns the environment and the method with all the shadows (except the execution shadows that are taken into account by the preShadowing function) wrapped by `impdep1` mnemonics and enrich the given shadows list by those new shadows. The function isInstShadow used returns true if the given instruction corresponds to a shadow. The wrapping starts from the given program counter:

shadowing:$Environment \times Method \times ProgramCounter \times (Shadow)$ *-list*$\rightarrow Environment \times Method \times (Shadow)$ *-list*

shadowing($\mathcal{E}$,$m$,$pc$,$l$) =shadowing($\mathcal{E}$,$m$,$pc$+1,$l$)

if $\neg$isInstShadow($\mathcal{E}$,$m$,$m.code(pc)$) $\wedge$ $pc \neq$ lastNumInst($m.code$)

shadowing($\mathcal{E}$,$m$,$pc$,$l$) =shadowing($\mathcal{E}'$,$m'$,$pc$+3, l::newShadow($\mathcal{E}$,$m$,$m.code(pc)$),pc,pc+2,[ ]))

if isInstShadow($\mathcal{E}$,$m$,$m.code(pc)$) $\wedge$ $pc \neq$ lastNumInst($m.code(pc)$)$\wedge$ insertImpdep1($\mathcal{E}$,$m$,$pc$)=($\mathcal{E}'$,$m'$)

shadowing($\mathcal{E}$,$m$,$pc$,$l$) =($\mathcal{E}$,$m$,$l$)if $\neg$isInstShadow($\mathcal{E}$,$m$,$m.code(pc)$)$\wedge$ $pc$=lastNumInst($m.code(pc)$)

shadowing($\mathcal{E}$,$m$,$pc$,$l$) =($\mathcal{E}'$,$m'$,l::newShadow(infoOf($\mathcal{E}$,$m$,$m.code(pc)$)),pc,pc+2.[ ]))

if isInstShadow($\mathcal{E}$,$m$,$m.code(pc)$) $\wedge$ $pc$=lastNumInst($m.code(pc)$)$\wedge$ insertImpdep1($\mathcal{E}$,$m$,$pc$)=($\mathcal{E}'$,$m'$)

38. The function signatureAspectOf returns the signature of the method "`aspectOf`" of the advice

aspect:

signatureAspectOf : *AdviceInfo* → *MethodSignature*

signatureAspectOf(*ad*) = *ms* if

$$\begin{cases} \text{ms.}name = \text{“}\texttt{aspectOf}\text{”}, \\ \wedge \text{ms.}argumentsType = [\,], \\ \wedge \text{ms.}resultType = \text{ad.}fromClass \end{cases}$$

39. The following function storeCode takes a shadow, a method, a list of types (initially argument types of the shadow) and a natural number that represents the current maximum length of the method local variables and returns a list of instructions. This maximum is needed in order to generate store instructions in order to put the shadow arguments amd its target on free places in the method local variables. If the shadow has its arguments on the stack ($\pi 2(s.kind) = \texttt{false}$), the function returns only store instructions in order to store the arguments and the target in temporary variables. If the shadow doesn't have its arguments on the stack, the function returns load and store instructions because we need to load the arguments from their original emplacement in the local variables table and restore them in temporary variables.

storeCode: *Method*×*Shadow*× (*Type*) `-list`× *Nat*→ (*Instruction*) `-list`

storeCode(*m*,*s*,[ ],*maxLocals*)=`astore_`*maxLocals*

$$\text{if} \begin{cases} \pi 2(s.kind) = \texttt{true} \\ \wedge \text{hasTarget(s,m)} \end{cases}$$

storeCode(*m*,*s*,[ ],*maxLocals*)=`aload_0::astore_`*maxLocals*

$$\text{if} \begin{cases} \pi 2(s.kind) = \texttt{false} \\ \wedge \text{hasTarget(s,m)} \end{cases}$$

storeCode(*m*,*s*,[ ],*maxLocals*)=[ ] if ¬ hasTarget(*s*,*m*)

storeCode($m,s,l::t,maxLocals$)= `istore_`$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if}\begin{cases}\text{isPrimitive}(t)\\ \wedge\pi 2(s.kind) = \texttt{true}\end{cases}$$

storeCode($m,s,l::t,maxLocals$)=`astore_`$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if}\begin{cases}\neg\text{isPrimitive}(t)\\ \wedge\pi 2(s.kind) = \texttt{true}\end{cases}$$

storeCode($m,s,l::t,maxLocals$)= `iload_`(length($l$)+1)::

`istore_`$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if}\begin{cases}\text{isPrimitive}(t)\\ \wedge\pi 2(s.kind) = \texttt{false}\\ \wedge\text{hasTarget}(s,m)\end{cases}$$

storeCode($m,s,l::t,maxLocals$)=`iload_`(length($l$))::

`istore_`$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if}\begin{cases}\text{isPrimitive}(t)\\ \wedge\pi 2(s.kind) = \texttt{false}\\ \wedge\neg\text{hasTarget}(s,m)\end{cases}$$

storeCode($m,s,l::t,maxLocals$)=`aload_`(length($l$)+1)::

`astore_`$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if}\begin{cases}\neg\text{isPrimitive}(t)\\ \wedge\pi 2(s.kind) = \texttt{false}\\ \wedge\text{hasTarget}(s,m)\end{cases}$$

storeCode($m,s,l::t,maxLocals$)=aload_(length($l$))::

astore_$maxLocals$::storeCode($m,s,l,maxLocals+1$)

$$\text{if} \begin{cases} \neg\text{isPrimitive}(t) \\ \wedge \pi 2(s.kind) = \texttt{false} \\ \wedge \neg\text{hasTarget}(s,m) \end{cases}$$

40. The function trans takes a branching JVML instruction, a natural number indicating the translation path in the the branching instruction in the code , and another number that indicates from which address, the translation begins:

trans : $JVMLInst \times Nat \times Nat \rightarrow JVMLInst$

trans($ins,i,pc$)=goto $adr'$ if

$$\begin{cases} ins = \texttt{goto}\ adr \\ \wedge((adr <= pc) \wedge (adr' = adr)) \vee ((adr > pc) \wedge (adr' = adr + i)) \end{cases}$$

trans($ins,i,pc$)=ifeq $adr'$ if

$$\begin{cases} ins = \texttt{ifeq}\ adr \\ \wedge((adr <= pc) \wedge (adr' = adr)) \vee ((adr > pc) \wedge (adr' = adr + i)) \end{cases}$$

trans($ins,i,pc$)=ifne $adr'$ if

$$\begin{cases} ins = \texttt{ifne}\ adr \\ \wedge((adr <= pc) \wedge (adr' = adr)) \vee ((adr > pc) \wedge (adr' = adr + i)) \end{cases}$$

41. The function translate takes a code, a natural number indicating the translation path in the the branching instructions in the code , and another number that indicates from which address, the translation begins:

translate : $Code \times Nat \times Nat \rightarrow Code$

translate($c,i,pc$)=$c'$ if

$c'(k)$=$c(k)$ $\forall$ k $\in$ Dom($c$) $\wedge$ $\neg$ branchIns($c(k)$)

$c'(k)$=trans($c(k),i,pc$) $\forall$ k $\in$ Dom($c$) $\wedge$ branchIns($c(k)$)

# APPENDIX III: λ_SAOP Semantics Utility Functions

1. The function containProceed takes around expression and returns true if this expression contains *proceed*. Otherwise, it returns false.

   containProceed : $ExpAr \rightarrow Boolean$

   containProceed($e'$)=

   $$\begin{cases} false \quad \text{if } e' = c \vee e' = x; \\ true \quad \text{if } e' = proceed; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \quad \text{if } e' = e_1' e_2'; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \quad \text{if } e' = e_1'; e_2'; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \quad \text{if } e' = \textbf{let rec } f\ x = e_1' \textbf{ in } e_2'; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \quad \text{if } e' = \textbf{let } x = e_1' \textbf{ in } e_2'; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \vee \text{containProceed}(e_3') \\ \text{if } e' = \textbf{if } e_1' \textbf{ then } e_2' \textbf{ else } e_3'; \\ \text{containProceed}(e_1') \quad \text{if } e' = \textbf{ref } e_1'; \\ \text{containProceed}(e_1') \quad \text{if } e' = !\ e_1'; \\ \text{containProceed}(e_1') \vee \text{containProceed}(e_2') \quad \text{if } e' = e_1' := e_2'. \end{cases}$$

2. The function $f_{app}$ takes an expression (the function name of an application expression), a type, a set of tags, and a sequence of defined advices and returns a sequence of applicable advices:

   $f_{app} : Exp \times Type \times TagSet \times AdvSeq \rightarrow AdvSeq$

   $f_{app}(e,\tau,t,s)$=

   $$\begin{cases} \varepsilon & \text{if } s = \varepsilon; \\ f_{app}(e,\tau,t,s') & \text{if } (s = as') \wedge \neg \text{matchJpCall}(e,\tau,t,a.pcd); \\ f_{app}(e,\tau,t,s') & \text{if } (s = as') \wedge \text{matchJpCall}(e,\tau,t,a.pcd). \end{cases}$$

3. The function $f_{assign}$ takes an expression (the left hand side of an assignment operator), a type, a set

of tags, and a sequence of defined advices and returns a sequence of applicable advices:

$\mathsf{f}_{assign} : Exp \times Type \times TagSet \times AdvSeq \rightarrow AdvSeq$

$\mathsf{f}_{assign}(e,\tau,t,s)=$

$$\begin{cases} \varepsilon & \text{if } s = \varepsilon; \\ \mathsf{f}_{assign}(e,\tau,t,s') & \text{if } (s = as') \wedge \neg \mathsf{matchJpSet}(e,\tau,t,a.pcd); \\ \mathsf{f}_{assign}(e,\tau,t,s') & \text{if } (s = as') \wedge \mathsf{matchJpSet}(e,\tau,t,a.pcd)\,. \end{cases}$$

4. The function $\mathsf{f}_{deref}$ takes an expression (the argument of a dereferencing operator), a type, a set of tags, and a sequence of defined advices and returns a sequence of applicable advices:

   $\mathsf{f}_{deref} : Exp \times Type \times TagSet \times AdvSeq \rightarrow AdvSeq$

   $\mathsf{f}_{deref}(e,\tau,t,s)=$

   $$\begin{cases} \varepsilon & \text{if } s = \varepsilon; \\ \mathsf{f}_{deref}(e,\tau,t,s') & \text{if } (s = as') \wedge \neg \mathsf{matchJpGet}(e,\tau,t,a.pcd)\,; \\ \mathsf{f}_{deref}(e,\tau,t,s') & \text{if } (s = as') \wedge \mathsf{matchJpGet}(e,\tau,t,a.pcd). \end{cases}$$

5. The function M takes a type and a mapping from region to a set of tags and returns a set of tags: $\mathsf{M} : Type \times (Region \xrightarrow[m]{} TagSet) \rightarrow TagSet$

   $\mathsf{M}(\tau,m)=$

   $$\begin{cases} m(\rho) & \text{if } \tau = ref_{\rho}(\tau') \wedge \rho \in Dom(m); \\ \{\} \texttt{ otherwise.} \end{cases}$$

6. The function matchJpCall takes an expression (the function name of an application expression), a type, a set of tags, and a pointcut and returns true if the pointcut attributes match the join point represented by the expression. Otherwise, it returns false:

   $\mathsf{matchJpCall} : Exp \times Type \times TagSet \times Pcd \rightarrow Boolean$

matchJpCall($e, \tau, t, p$)=

$$\begin{cases} \text{matchJpCall}(e,\tau,t,p1) \wedge \text{matchJpCall}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\ \text{matchJpCall}(e,\tau,t,p_1) \vee \text{matchJpCall}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\ \neg\text{matchJpCall}(e,\tau,t,p_1) & \text{if } p = \neg p_1; \\ \textit{true} & \text{if } p.pkind = \texttt{call} \wedge p.var = e \wedge p.typeScheme \succ \tau; \\ \textit{true} & \text{if } p.pkind = \texttt{dflow} \wedge \text{matchJpCall}(e,\tau,t,p.pcd_2) \wedge \ p.tag \in t; \\ \textit{false} & \texttt{otherwise}. \end{cases}$$

7. The function matchJpGet takes an expression (the argument of a dereferencing operator), a type, a set of tags, and a pointcut and returns true if the pointcut attributes match the join point represented by the expression. Otherwise, it returns false:

matchJpGet : $Exp \times Type \times TagSet \times Pcd \rightarrow Boolean$

matchJpGet($e, \tau, t, p$)=

$$\begin{cases} \text{matchJpGet}(e,\tau,t,p_1) \wedge \text{matchJpGet}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\ \text{matchJpGet}(e,\tau,t,p_1) \vee \text{matchJpGet}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\ \neg\text{matchJpGet}(e,\tau,t,p_1) & \text{if } p = \neg p_1; \\ \textit{true} & \text{if } p.pkind = \texttt{get} \wedge p.var = e \wedge p.typeScheme \succ \tau; \\ \textit{true} & \text{if } p.pkind = \texttt{dflow} \wedge \text{matchJpGet}(e,\tau,t,p.pcd_2) \wedge \\ & \ p.tag \in t; \\ \textit{false} & \texttt{otherwise}. \end{cases}$$

8. The function matchJpSet takes an expression (the left hand side of an assignment operator), a type, a set of tags, and a pointcut and returns true if the pointcut attributes match the join point represented by the expression. Otherwise, it returns false:

matchJpSet : $Exp \times Type \times TagSet \times Pcd \rightarrow Boolean$

matchJpSet($e, \tau, t, p$)=

$$\begin{cases} \mathsf{matchJpSet}(e,\tau,t,p_1) \wedge \mathsf{matchJpSet}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\ \mathsf{matchJpSet}(e,\tau,t,p_1) \vee \mathsf{matchJpSet}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\ \neg\mathsf{matchJpSet}(e,\tau,t,p_1) & \text{if } p = \neg\, p_1; \\ \mathit{true} & \text{if } p.pkind = \texttt{set} \wedge p.var = e \wedge p.typeScheme \succ \tau; \\ \mathit{true} & \text{if } p.pkind = \texttt{dflow} \wedge \mathsf{matchJpSet}(e,\tau,t,p.pcd_2) \\ p.tag \in t; & \\ \mathit{false} & \texttt{otherwise}. \end{cases}$$

9. The function searchTagCall takes an expression (the function name of an application expression), a type, a set of tags, and a sequence of defined advices and returns a set of tags:

   $\mathsf{searchTagCall} : \mathit{Exp} \times \mathit{Type} \times \mathit{TagSet} \times \mathit{AdvSeq} \rightarrow \mathit{TagSet}$

   $\mathsf{searchTagCall}(e,\tau,t,s)=$

   $$\begin{cases} \{\} & \text{if } s = \varepsilon; \\ \mathsf{searchTagCallPcd}(e,\tau,t,a.pcd) \cup \mathsf{searchTagCall}(e,\tau,t,s') & \text{if } (s = as'). \end{cases}$$

10. The function searchTagCallPcd takes an expression (the function name of an application expression), a type, a set of tags, and a pointcut and returns a set of tags:

    $\mathsf{searchTagCallPcd} : \mathit{Exp} \times \mathit{Type} \times \mathit{TagSet} \times \mathit{Pcd} \rightarrow \mathit{TagSet}$

    $\mathsf{searchTagCallPcd}(e,\tau,t,p)=$

$$\begin{cases} \text{searchTagCallPcd}(e,\tau,t,p_1) \cup \text{searchTagCallPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\ \text{searchTagCallPcd}(e,\tau,t,p_1) \cup \text{searchTagCallPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\ \text{searchTagCallPcd}(e,\tau,t,p_1) & \text{if } p = \neg\, p_1; \\ \{\} & \text{if } p.kind <> \texttt{dflow}; \\ \{p.tag\} & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpCall}(e,\tau,t,p.pcd_1) \wedge \\ & \quad p.pcd_1.kind <> \texttt{dflow}; \\ \{p.tag\} \cup \text{searchTagCallPcd}(e,\tau,t,p.pcd_1) & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpCall}(e,\tau,t,p.pcd_1) \wedge \\ & \quad p.pcd_1.kind = \texttt{dflow}. \end{cases}$$

11. The function searchTagGet takes an expression (the argument of a dereferencing operator), a type, a set of tags, and a sequence of defined advices and returns a set of tags:

searchTagGet : $Exp \times Type \times TagSet \times AdvSeq \rightarrow TagSet$

searchTagGet$(e,\tau,t,s)$=

$$\begin{cases} \{\} & \text{if } s = \varepsilon; \\ \text{searchTagGetPcd}(e,\tau,t,a.pcd) \cup \text{searchTagGet}(e,\tau,t,s') & \text{if } (s = as'). \end{cases}$$

12. The function searchTagGetPcd takes an expression (the argument of a dereferencing operator), a type, a set of tags, and a pointcut and returns a set of tags:

searchTagGetPcd : $Exp \times Type \times TagSet \times Pcd \rightarrow TagSet$

searchTagGetPcd$(e,\tau,t,p)$=

$$\begin{cases} \text{searchTagGetPcd}(e,\tau,t,p_1) \cup \text{searchTagGetPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\ \text{searchTagGetPcd}(e,\tau,t,p_1) \cup \text{searchTagGetPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\ \text{searchTagGetPcd}(e,\tau,t,p_1) & \text{if } p = \neg\, p_1; \\ \{\} & \text{if } p.kind <> \texttt{dflow}\,; \\ \{p.tag\} & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpGet}(e,\tau,t,p.pcd_1) \wedge \\ & \quad p.pcd_1.kind <> \texttt{dflow}; \\ \{p.tag\} \cup \text{searchTagGetPcd}(e,\tau,t,p.pcd_1) & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpGet}(e,\tau,t,p.pcd_1) \wedge \\ & \quad p.pcd_1.kind = \texttt{dflow}. \end{cases}$$

13. The function searchTagSet takes an expression (the left hand side of an assignment operator), a type, a set of tags, and a sequence of defined advices and returns a set of tags:

searchTagSet : $Exp \times Type \times TagSet \times AdvSeq \rightarrow TagSet$

searchTagSet$(e,\tau,t,s)$=

$$\begin{cases} \{\} & \text{if } s = \varepsilon; \\ \text{searchTagSetPcd}(e,\tau,t,a.pcd) \cup \text{searchTagSet}(e,\tau,t,s') & \text{if } (s = as'). \end{cases}$$

14. The function searchTagSetPcd takes an expression (the left hand side of an assignment operator), a type, a set of tags, and a pointcut and returns a set of tags:

searchTagSetPcd : $Exp \times Type \times TagSet \times Pcd \rightarrow TagSet$

searchTagSetPcd$(e,\tau,t,p)$=

$$
\begin{cases}
\text{searchTagSetPcd}(e,\tau,t,p_1) \cup \text{searchTagSetPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \wedge p_2; \\
\text{searchTagSetPcd}(e,\tau,t,p_1) \cup \text{searchTagSetPcd}(e,\tau,t,p_2) & \text{if } p = p_1 \vee p_2; \\
\text{searchTagSetPcd}(e,\tau,t,p_1) & \text{if } p = \neg\, p_1; \\
\{\} & \text{if } p.kind <> \texttt{dflow}\,; \\
\{p.tag\} & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpSet}(e,\tau,t,p.pcd_1) \wedge \\
\quad p.pcd_1.kind <> \texttt{dflow}; & \\
\{p.tag\} \cup \text{searchTagSetPcd}(e,\tau,t,p.pcd_1) & \text{if } p.kind = \texttt{dflow} \wedge \text{matchJpSet}(e,\tau,t,p.pcd_1) \wedge \\
\quad p.pcd_1.kind = \texttt{dflow}. &
\end{cases}
$$

15. The function TypeOf takes a constant and returns its type scheme:

TypeOf : *Const* → *TypeScheme*

TypeOf($c$)= *int* if $c = \mathbf{n}$

TypeOf($c$)=*unit* if $c =$ **( )**

TypeOf($c$)=*bool* if $c =$ **true** $\vee$ $c =$ **false**