

# Flow logic for language-based safety and security

**Hansen, René Rydhof**

*Publication date:*
2005

*Document Version*
Publisher's PDF, also known as Version of record

# Flow Logic for Language-Based Safety and Security

René Rydhof Hansen

Kongens Lyngby 2005
IMM-PHD-2005-143

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

# Summary

Society is increasingly dependent on information and communication technology. Computers are integrated into everything from toasters to control systems for critical infrastructure. Consequently even simple programming errors have the potential to wreck havoc on practically every aspect of society and everyday life. It is therefore a crucial challenge for computer science to develop tools and techniques that can help improve the quality of software design and implementation.

In this dissertation it is argued that techniques rooted in the theory and practice of programming languages, so-called *language-based safety and security*, provide a feasible platform for developing software that can be verified and validated with a very high degree of assurance. Specifically, it is argued and demonstrated that *static analysis* is an indispensable technique for language-based safety and security and that the *Flow Logic* framework for static analysis is particularly useful this regard.

In order to support and illustrate the above points, a number of program analyses are developed for the Carmel programming language, a variant of the low-level language used on Java smart cards. The analyses are formally proved correct with respect to the semantics and are then used to verify a wide spectrum of pertinent safety and security properties.

# Resumé

Samfundet afhænger i stadig stigende grad af informations- og kommunikationsteknologi. Computere indbygges i alt fra brødristere til kontrolsystemer for kritisk infrastruktur. Derfor kan selv simple programmeringsfejl have en uoverskuelig ødelæggende effekt på alle aspekter af samfundet. Det er derfor en grundlæggende udfordring for datalogien at udvikle værktøjer og teknikker der kan bruges til at forbedre kvaliteten af den software der i fremtiden skal designes og implementeres.

I denne afhandling argumenteres der for at teknikker der har rod i teorien om programmeringssprog, såkaldt *sprogbaseret sikkerhed*, udgør en nyttig platform for udvikling af programmer der med en høj grad af sikkerhed kan verificeres og valideres. Specifikt argumenteres der for at *statisk analyse* er en uundværlig metode i relation til sprogbaseret sikkerhed og at *Flow Logic* er særligt brugbart i den forbindelse.

For at understøtte og illustrere ovenstående pointer, udvikles der et antal programanalyser for programmeringssproget Carmel, der er en variant af det lavniveau-sprog der anvendes på Java smart cards. Analyserne bevises formelt korrekte med hensyn til semantikken og bruges derefter til at verificere et bredt spektrum af sikkerhedsegenskaber.

# Preface

*Problems worthy of attack*
*prove their worth by hitting back*
*—Piet Hein*

This dissertation was written at the Technical University of Denmark, Informatics and Mathematical Modelling. Supervised by Flemming Nielson. The work reported in the following chapters grew out of the SecSafe project, cf. [Siv03b], whose goal it was to investigate the use of static analysis for verifying security properties of Java Card bytecode programs.

The basic control flow analysis, exception analysis, data flow analysis, and ownership analysis as well as a preliminary report on implementation issues are published as SecSafe technical reports: [Han02c, Han02b, Han02d]. The ownership analysis and a prototype implementation used for firewall validation is published in [Han02a]. Transaction flow analysis for Carmel (with exceptions) is published in [HS05] and the devil's advocate in [Han04].

**Acknowledgements.** First of all I would like to thank Flemming Nielson, my supervisor, and Hanne Riis Nielson for their enduring support. Thanks also to Mikael Buchholtz for our many talks and discussions: always interesting, sometimes relevant, and never boring; to Igor A. Siveroni for putting up with my many questions and for his hospitality; and to my fellow students and the staff at IMM for creating a pleasant atmosphere. Jacob Grydholt Jensen deserves special thanks for reading and commenting on an early draft. Finally I owe my wife, family, and friends a debt of gratitude for their support and patience.

René Rydhof Hansen
Kongens Lyngby, January 2005

# Contents

Chapter 1

# Introduction

*When you think how well basic appliances work,*
*it's hard to believe anyone ever gets on an airplane*
*—Calvin (Calvin & Hobbes)*

Relying on computers to perform critical tasks is fraught with danger and may have dire consequences. Lethal consequences even. This is the inevitable conclusion reached from even a cursory reading of the ACM Forum on Risks to the Public in Computers and Related Systems (known as "the RISKS forum"), cf. [RIS, Neu95]. Moreover, in many of the incidents discussed in the RISKS forum, faulty software is either the direct cause of or a contributing factor to the incident. With this less than impressive history of software reliability in mind, the stated goal of pervasive, or ubiquitous, computing to provide "information and communication technology everywhere, for everyone, at all times", cf. [CfP04], suddenly seems like a dangerous proposition. While several methodologies, tools, and techniques exist for developing trustworthy and reliable systems, they often add considerable overhead to the development process both in terms of time spent and resources expended. In addition, the deployment of formal methods for developing reliable systems often require specialist tools and training that may be hard and expensive to come by. For these reasons such methodologies have mainly found use in the development of high-assurance systems where the extra time and cost can be justified. However, if the full potential of modern information and communications technology is to be realised, tools and techniques for increasing software quality that are more accessible to non-specialists and more easily integrated into the design and implementation processes must be developed and employed.

In recent years techniques from programming languages, e.g., static analysis, type systems, and program rewriting, have been successfully used for verifying and validating safety and security properties of programs and has been ad-

vanced as a promising partial solution to some of the problems outlined above, cf. [Ame02, Bar03, Koz99, SMH01, Nec97]. The language-based approach is particularly appealing for several reasons: often concepts already familiar to programmers are leveraged, e.g., type systems, making it easier for programmers to understand and use; being rooted in programming languages means that there is a long tradition for implementing highly automated tools that can be integrated into existing development tools, e.g., compilers and static checkers.

The work in this dissertation supports the above arguments by developing Flow Logic based static analyses that are used to verify a number of safety and security properties of a stack-based low-level bytecode language based on the Java Card Virtual Machine Language, cf. [Sun00, Che00]. Using Java Card bytecode as basis for this dissertation has several advantages. First, it demonstrates how the Flow Logic framework copes with both low-level languages and advanced high-level language features such as objects and exceptions. Second, it minimises the *trusted computing base* for high-assurance applet by eliminating the need to trust (or prove) that a given Java Card compiler produces correct code. This is particularly important when developing applications that must be certified to comply with a given security standard, e.g., the Common Criteria [CC99]. Finally, low-level bytecode languages have already seen widespread use in embedded systems, notably smart cards, and this trend can be expected to continue. Therefore it is important to gain an understanding of how to develop high-quality and high-assurance applications for such systems.

## 1.1 Thesis

The main thesis of the dissertation can be formulated as follows:

> *The Flow Logic framework for static analysis is a powerful tool for language-based safety and security.*

In order to support this thesis the Flow Logic framework must be shown to have, at least, the following properties:

- Versatility: the framework must support many different notions of computation as well as unique and non-standard language features.
- Flexibility: it must support the development and specification of many different types of analyses and properties.
- Robustness: minor changes and additions to both the analysis specification and the underlying semantics should be easily accommodated in the framework.
- Scalability: the framework should be able to cope with all the features of a full programming language.
- Implementability: the framework must support a clear implementation strategy.

In this dissertation it is argued that the Flow Logic framework does indeed have all of the above properties. The emphasis will be on showing versatility (in terms of non-standard language features), flexibility, robustness, and scalability since it has already been demonstrated that the Flow Logic framework supports many different models of computation, e.g., [BBD+03, NHN03, BDNN01b, BDNN01a, NN98a, NN97, GNN97], and that it supports implementation through the systematic development of constraint generators over suitable constraint languages, e.g., the alternation-free least fixed-point logic [NNS+04, BBD+03, NNS02b, NNS02a, BNN02, NS01, Pil03, Han02d, Han02a].

In this dissertation the above thesis is examined and proved by developing a number of analyses for a low-level language based on Java Card bytecode. The analyses are developed in a staged, or modular, fashion and then used to verify safety and security properties, some of which are non-standard and unique to the Java Card platform. Details are given in the next section. As a consequence of this work, it will also be argued that language-based safety and security is a compelling methodology for developing safe and secure programs, and that static analysis is an indispensable technique for language-based safety and security.

## 1.2 Safety and Security on a Smart Card

For many safety and security critical applications smart cards provide an ideal platform due to the physical properties of the cards that make it very hard to change programs or data on the card, at least without leaving very visible evidence; this feature of smart cards is called "tamper-resistance" or "tamper-evident". Smart cards have, traditionally, been programmed using proprietary languages and programming tools making it very hard to port applications between different brands of smart cards. In addition programmers had to be trained for particular smart cards. This changed with the advent of the Java Card platform which is essentially a smart card with a built-in Java Card Virtual Machine (JCVM). The JCVM implements a subset of the Java Virtual Machine but extended with special features for smart card support, e.g., an applet[1] firewall mechanism. The JCVM is an interpreter for the Java Card Virtual Machine Language (JCVML), a stack-based low-level bytecode language similar to the Java bytecode language, most often produced by compiling programs written in Java Card, an extended subset of Java that only includes rudimentary language constructs and simple data types from Java proper. A short history of the Java Card platform and related languages can be found in [Che00].

The unpredictable environment in which Java Cards are used, e.g., a card may be torn from the card reader at any point during computation, and the somewhat esoteric protocols and coding conventions used in Java Card programming conspire to make it difficult to guarantee that a program and its data are handled in a sufficiently safe and secure manner. The EU research project SecSafe, cf. [Siv03b], was conceived to address this problem through the use of

[1] Java Card applications are traditionally referred to as *applets*.

static analysis for verification and validation. As a part of that project, a list of specific problems of immediate interest was identified as a possible starting point, cf. [MM01]. A subset of the identified problems are summarised below:

1. Information Flow Control: smart card programs often manipulate confidential data, e.g., PIN codes and personal information, it is important to ensure that such information is not leaked.

2. Service Control: certain functions of an applet should only be accessible under specific conditions, e.g., card reset should only be possible in special card terminals.

3. Error Prediction: only specific exceptions should be allowed to reach the top-level, i.e., the user.

4. Atomic Updates: certain sets of operations should be performed in the same transaction.

5. Overflow Control: numerical operations in Java Card may overflow silently resulting in errors and potential breach of security.

The problems of Information Flow Control and Error Prediction are targeted directly by the analyses developed in this dissertation, while Service Control, Atomic Updates, and Overflow Control are partially handled or could be handled by straightforward extensions of the pertinent analyses.

## 1.3 Outline of the Dissertation

In Chapter 2 the Carmel language is introduced as a rational reconstruction of the Java Card Virtual Machine Language. The Carmel language will serve as a vehicle of exposition throughout the dissertation.

In Chapter 3 a control flow analysis of Carmel is developed and proved correct. The control flow analysis is at the heart of all the subsequent analyses. An implementation of the control flow analysis, as a constraint generator over the alternation-free least fixed-point logic, is discussed in details and also formally proved correct. Finally, the control flow analysis is extended in a natural way to encompass an exception analysis. The correctness proof for the control flow analysis is extended in a similarly simple and natural way to cover the entire exception analysis. The developments in this chapter mainly shows robustness, scalability, and implementability but also to some extent versatility.

Chapter 4 further examines the versatility, robustness and flexibility of the Flow Logic framework by developing two analyses, a data flow analysis and an ownership analysis respectively, as natural extensions of the control flow and exception analyses. The former analysis illustrates how traditional analyses can be specified and integrated. The latter analysis shows how features that are unique to the Java Card Virtual Machine Language can be modelled and analysed.

Two advanced analyses are discussed in Chapter 5. The first is a transaction flow analysis that is used to verify that the use of transactions in a given program does not give rise to any errors. In contrast to the previous analyses, the transaction flow analysis is a context dependent analysis and thus also serves to demonstrate how the framework copes with such analyses. The second analysis in this chapter is an information flow analysis that is used to ensure that confidential data does not leak. This is established by showing that the analysis can be used to guarantee a non-interference property. Whereas the previous analyses are all first-order analyses, i.e., abstract properties describe (sets of) concrete values, the information flow analysis is a second-order analysis, i.e., abstract properties represent relations between values. This gives rise to a more complicated and involved proof of correctness. This work demonstrates a high degree of versatility and flexibility of the framework.

An important aspect of the Java Card platform is that it allows for programs to be downloaded dynamically and thereby extend the runtime environment. Chapter 6 defines an abstract representation of the analysis result of all possible dynamic extensions. This representation is called the *devil's advocate* and by analysing a program in conjunction with the devil's advocate it is possible to guarantee that the program will not leak certain information to any program later downloaded onto the card. In essence it enables analyses to work even under an "open world" assumption. The concept of devil's advocate developed in this chapter offers a very compelling illustration of the flexibility of the Flow Logic framework.

CHAPTER 2

# The Carmel Language

*A scientific theory should be as simple as possible, but no simpler*
*—Albert Einstein*

In this chapter the *Carmel* language is introduced. Carmel will serve as a vehicle of exposition and as a target for the problems, properties, and analyses of the following chapters. As such, the intention is for Carmel to provide a basis for exposition and discussion; consequently the definition of Carmel emphasises simplicity and succinctness over features and efficiency. To carry this intention even further a small subset of Carmel, called Carmel Core, is identified and used as the theoretical basis for the analyses and developments in chapters 5 and 6.

In Section 2.1 the design rationale and background for the Carmel language and its variants are discussed in more detail while Carmel Core is defined in Section 2.6.

## 2.1 From Java Card Bytecode to Carmel

Carmel is the descendant of a language called JCVML$_e$ that was originally conceived as the target of study for the SecSafe project, cf. [Siv03b, Siv04, Mar01, SH01], that investigated the application of program analysis to ensure the safety and security of Java Card systems. The idea was to create a language that was based on and as expressible and powerful as Java Card Virtual Machine Language[1] (JCVML), but targeted more towards formal methods and proofs. This resulted in a language with 31 instructions as opposed to the more than 180 instructions of JCVML and where names and types were used directly instead of the corresponding tokens and offsets used in JCVML. This made JCVML$_e$

[1] Also known as Java Card Bytecode

well-suited for formal methods and, equally important, for communicating ideas and analyses.

Even though JCVML$_e$ represents a significant simplification, or rationalisation, of JCVML it retains all the expressive power of JCVML. Indeed a trivial syntactic substitution suffices to translate a JCVML program into the corresponding JCVML$_e$ program. The name *Carmel* was later adopted and used for various subsets, incarnations, and variations of JCVML$_e$. The program structure and rationale for JCVML$_e$ (and later Carmel) is described in [Mar01]. An operational semantics is given in [Siv04, SH01]. The underlying runtime system, the Java Card Runtime Environment (JCRE), is formalised in [Siv03a] along with parts of the API supported by the JCRE. Program structure and operational semantics for the Carmel variant studied in this dissertation can be found in sections 2.2 and 2.4 respectively. The semantics of [Siv04, SH01] define not only a semantics for the JCVML$_e$ language but also for parts of the underlying runtime-environment as defined in [Sun00]. Most notably the notion of *ownership* and the related *firewall* of the Java Card Runtime Environment, cf. [SJE01], since these are inextricably linked to the execution of a program on a Java smart card.

In this dissertation the goal is not to define one base language that fully captures all the special features of JCVML and the JCRE. The goal is to define a target language that is as small and simple as possible while still being useful for the intended purpose. Any extra features of interest, e.g., the above mentioned firewall, can then be added to the base language and studied as needed. This is a quite common approach when studying programming languages: to have a small base language for theoretical studies and then a larger language more directly oriented towards application. Such an approach has several advantages: it reduces the overhead when adding new features and it provides a clearer picture of how various language components interact. As a further step in this direction a small subset of Carmel proper is identified that includes only the most basic and important instructions and features. This subset, called Carmel Core and described in more detail in Section 2.6, will serve as the base language for the theoretical developments in chapters 5 and 6.

## 2.2 Program Structure

The formal structure for Carmel programs is defined and discussed in this section. Following the approach of [Mar01] an abstract foundation is laid for the Carmel language in which programs are represented as an abstract data structure that is accessed through a number of special *access functions*.

In contrast to a more traditional approach of representing programs as (abstract) syntax trees the data structure approach has the advantage that it is very easy to extract relevant information from a program in a very convenient manner. The disadvantage, however, is that the abstract data structure does not convey the actual program structure in a very visual or intuitive manner. Therefore a more readable syntax, based on the Java Card syntax, for actual

| | | |
|---|---|---|
| Type | ::= | RefType \| PrimType |
| PrimType | ::= | `boolean` \| `byte` \| `short` \| `int` |
| RefType | ::= | ArrayType \| SimpleRef |
| SimpleRef | ::= | ClassName \| InterfaceName |
| ElemType | ::= | SimpleRef \| PrimType |
| ArrayType | ::= | (`array` ElemType) |
| ReturnType | ::= | Type \| `void` |
| MethodType | ::= | $\mathsf{Type}^* \rightarrow \mathsf{ReturnType}$ |

Figure 2.1: Carmel types

program examples is briefly discussed in Section 2.2.11.

### 2.2.1 Notation

Following [Mar01, Siv04] a *domain* is defined to be a set equipped with corresponding *access functions* that are used to access and modify various components of the domain. The *record* notation is also introduced as a particularly convenient way to specify a domain with all its access functions:

$$\mathsf{Dom} = (f_1 : \mathsf{Dom}_1) \times \ldots \times (f_n : \mathsf{Dom}_n)$$

which defines the domain, Dom, with access functions $f_i : \mathsf{Dom} \rightarrow \mathsf{Dom}_i$ for $1 \leq i \leq n$. Access to an element, $f_i$, of $d \in \mathsf{Dom}$ is written in an "object-oriented" style: $d.f_i$ and similarly updating an element is written $d[f_i \mapsto v]$. Lifting the domain Dom to a domain that includes a (new) bottom value is written $\mathsf{Dom}_\perp$.

To enhance legibility and reduce the number of parentheses the above notation is extended from access functions to functions in general: writing $x.f$ for $f(x)$ and $x_1.f(x_2, \ldots, x_n)$ for $f(x_1, x_2, \ldots, x_n)$.

### 2.2.2 Types

Carmel (and JCVML) is a strongly typed language and type declarations are an intrinsic part of the program structure. In Figure 2.1 the Carmel types are shown. Note that unlike the semantics of [Siv04, SH01] the Carmel semantics presented here does not model word sizes or word boundaries and thus does not need to distinguish between the primitive types; in essence all of the primitive types, denoted PrimType, are considered simply as numbers. For the analyses and properties discussed in the following chapters this memory model is sufficient.

Figure 2.2 defines a subtype relation, denoted $\preceq$, on the Carmel types. The functions $\mathit{super}^*$ (for both classes and interfaces) and $\mathit{implements}^*$ are defined in Sections 2.2.5 and 2.2.6.

$$\frac{t \in \mathsf{ElemType}}{t \preceq t} \qquad \frac{t \preceq t'}{(\texttt{array}\ t) \preceq (\texttt{array}\ t')}$$

$$\frac{\sigma' \in super^*(\sigma)}{\sigma \preceq \sigma'} \qquad \frac{iface' \in super^*(iface)}{iface \preceq iface'}$$

$$iface \preceq \texttt{Object} \qquad \frac{iface \in implements^*(\sigma)}{\sigma \preceq iface}$$

Figure 2.2: Subtyping relation

### 2.2.3 Programs

In order to facilitate the co-existence of several applets on a Java Card, JCVML programs are made up of a number of *packages* that introduce separate namespaces and thus can be thought of as programs in their own right.

For Carmel a similar structure is adopted and a *program* is defined to be a set of packages:

$$\mathsf{Program} = (packages : \mathcal{P}(\mathsf{Package}))$$

### 2.2.4 Packages

Each package has a name and contains the set of classes and interfaces defined in it:

$$\begin{aligned}\mathsf{Package} = {} & (name : \mathsf{PackageName}) \times \\ & (classes : \mathcal{P}(\mathsf{Class})) \times \\ & (interfaces : \mathcal{P}(\mathsf{Interface}))\end{aligned}$$

Since packages are uniquely identified by their name (in a given program) a package may be unambiguously referenced simply by its name rather than its full definition.

### 2.2.5 Classes

Classes in Carmel, as in other class-based object oriented languages, provide the main abstraction mechanism of the language. They are instantiated as objects that capture state information and the methods for manipulating the state in a natural manner.

A class is defined by the package it belongs to, its name, and its place in the class hierarchy. The class hierarchy is encoded in the $super : \mathsf{Class} \rightarrow \mathsf{Class}_\perp$ function that returns the superclass of the given class. In keeping with JCVML the existence of a special class, `Object`, is assumed that is implicitly the superclass of all classes except itself; thus the *super* function will return the bottom value, $\perp$, when used on the `Object` class. The methods and fields defined by a class are accessed through the *methods* and *fields* component respectively.

Finally, a class may implement a number of *interfaces*, as defined in the next section, available through the *implements* function:

$$\begin{aligned}\mathsf{Class} = {} & (\mathit{name} : \mathsf{ClassName}) \times \\ & (\mathit{package} : \mathsf{Package}) \times \\ & (\mathit{super} : \mathsf{Class}_{\perp}) \times \\ & (\mathit{methods} : \mathcal{P}(\mathsf{Method})) \times \\ & (\mathit{fields} : \mathcal{P}(\mathsf{Field})) \times \\ & (\mathit{implements} : \mathcal{P}(\mathsf{Interface}))\end{aligned}$$

A class inherits all the static and instance fields of its super class. Similarly static and virtual methods are inherited from the super class. In addition to implementing new static and virtual methods a class may supply its own implementation of a virtual method already defined in the super class and thereby *override* the previous definition. When a virtual method is invoked a *method lookup* is performed at runtime to search the class hierarchy for the applicable implementation of the invoked method. This process is also called *dynamic method dispatch*. The method lookup is formally defined in conjunction with the semantics of method invocation. In Chapter 3 a control flow analysis is defined that, among other things, computes an over-approximation of the method implementations that may be used at each invocation point in a program.

Classes may be referred to using the *qualified name* of the class, i.e., the combination of package name and class name which uniquely identifies a class in a given program. As an example consider the class named `C` belonging to the package `p`, this class is referred to as "`p.C`". When no confusion can arise the package name may be elided, e.g., when a program only contains one package.

The following two functions are defined for later use. The first determines the set of super classes for a given class by transitively traversing the class hierarchy:

$$\begin{aligned}\mathit{super}^*(\perp) &= \emptyset \\ \mathit{super}^*(\sigma) &= \{\sigma.\mathit{super}\} \cup (\sigma.\mathit{super}).\mathit{super}^*\end{aligned}$$

The next determines the set of interfaces implemented (transitively) by a class by first taking the union of all the interfaces directly implemented by the class ($\sigma.\mathit{implements}$) and all the super-interfaces of those (($\sigma.\mathit{implements}$).$\mathit{super}^*$), and finally recursively add all the interfaces implemented by the superclass (($\sigma.\mathit{super}$).$\mathit{implements}^*$):

$$\begin{aligned}\mathit{implements}^*(\perp) &= \emptyset \\ \mathit{implements}^*(\sigma) &= \\ & \sigma.\mathit{implements} \cup (\sigma.\mathit{implements}).\mathit{super}^* \cup (\sigma.\mathit{super}).\mathit{implements}^*\end{aligned}$$

The $\mathit{super}^*$ for interfaces is defined in the next section.

### 2.2.6 Interfaces

Since JCVML, and consequently Carmel, does not allow a class to inherit from more than one class *interfaces* can be seen as a way to regain some of the advan-

tages of multiple-inheritance and yet avoid the problems inherent in multiple-inheritance.

$$\begin{aligned}\mathsf{Interface} = {} & (\mathit{name} : \mathsf{InterfaceName}) \times \\ & (\mathit{package} : \mathsf{Package}) \times \\ & (\mathit{super} : \mathcal{P}(\mathsf{Interface})) \times \\ & (\mathit{methods} : \mathcal{P}(\mathsf{Method})) \times \\ & (\mathit{fields} : \mathcal{P}(\mathsf{Field})) \times \\ & (\mathit{implementedBy} : \mathcal{P}(\mathsf{Class}))\end{aligned}$$

The methods of an interface are called *abstract* methods and do not contain any instructions. A class that *implements* a given interface must provide implementations of *all* the methods declared in the interface. Interfaces behave like classes regarding fields and methods: fields are inherited and methods are looked up.

Interfaces may be referred to using only the qualified name of the interface and as for classes, the package name may be omitted when it is unambiguously defined by context.

The following function transitively computes the complete set of super-interfaces to a given interface. For convenience it is defined on a set of interfaces:

$$\begin{aligned}& \mathit{super}^*(\{\mathit{iface}_1, \ldots, \mathit{iface}_n\}) = \\ & \quad \textstyle\bigcup_{1 \le i \le n} \mathit{iface}_i.\mathit{super} \\ & \quad \textstyle\bigcup_{1 \le i \le n} \{\mathit{iface} \mid \mathit{iface} \in \mathit{iface}_i.\mathit{super}.\mathit{super}^*\}\end{aligned}$$

### 2.2.7 Methods

The actual functionality of a Carmel program is implemented by *methods*. Methods are identified by their name, the class or interface in which they are defined, and their type. Furthermore methods contain the instructions that actually implements the method, accessible through the function *instructionAt* that takes a program counter and returns the corresponding instruction or bottom if there is no instruction at that program counter. Finally, a method can be either a *static method* or a *virtual method* as indicated by *isStatic*:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{class} : \mathsf{Class} \cup \mathsf{Interface}) \times \\ & (\mathit{name} : \mathsf{MethodName}) \times \\ & (\mathit{type} : \mathsf{MethodType}) \times \\ & (\mathit{instructionAt} : \mathsf{PC} \to \mathsf{Instruction}_\perp) \times \\ & (\mathit{isStatic} : \mathsf{Bool}) \\ & (\mathit{handlers} : \mathbb{N}_0 \to \mathsf{ExcHandler}_\perp)\end{aligned}$$

The first instruction of a method is located at program counter 0 and for a non-branching instruction at program counter $pc$ the next instruction can be found at program counter $pc + 1$. For $m.\mathit{type} = (t_1 :: \cdots :: t_n) \to t'$ define $m.\mathit{returnType} = t'$.

A method may also define a number of *exception handlers* accessed through the *handlers* component. The ordering of handlers, as defined by the argument

to the *handlers* function, determines which handler to use in the case of overlapping handlers. It is the responsibility of the compiler to define a suitable ordering as specified in [LY99]. The *handlers* function returns the bottom value for indices that has not been assigned to a handler.

Methods that are defined in interfaces are called *abstract methods* and do not contain any instructions. In contrast a class that implements a given interface must provide actual the implementations of all the methods defined in the interface.

The qualified name of a method is not sufficient to uniquely identify a given method since Carmel (and JCVML) allows a form of *overloading* where different methods can have the same name provided they have different argument types. This implies that a method can be identified by using its argument type in addition to the qualified name. For example: `p.C2.abs(p.Num)` identifies the method named `abs` that takes a (reference to a) `p.Num` as argument and is defined in the class `C2` which belongs to the package `p`. For methods that are not overloaded the argument type may also be omitted.

### 2.2.8 Exception Handlers

Every method can define a number of local exception handlers to catch specific exceptions that are thrown in a specified region of the method. Thus exception handlers are composed as follows:

$$\begin{aligned}\mathsf{ExcHandler} = {} & (\mathit{catchType} : \mathsf{Class}_{\perp})\times \\ & (\mathit{startAddr} : \mathsf{PC})\times \\ & (\mathit{endAddr} : \mathsf{PC})\times \\ & (\mathit{handlerAddr} : \mathsf{PC})\end{aligned}$$

where *catchType* determines which exception to catch (with $\perp$ indicating any exception) while *startAddr* and *endAddr* specifies the region in which the exception should be caught and finally the start address of the handler itself is given by *handlerAddr*.

Note that it is the responsibility of the compiler to define an order on exception handlers so that the innermost (relevant) handlers appear first as specified in [LY99, Sun00].

### 2.2.9 Fields

Fields are given by the combination of the class in which they are define, the name of the field, and the type of the field:

$$\begin{aligned}\mathsf{Field} = {} & (\mathit{class} : \mathsf{Class} \cup \mathsf{Interface})\times \\ & (\mathit{name} : \mathsf{FieldName})\times \\ & (\mathit{type} : \mathsf{Type})\times \\ & (\mathit{isStatic} : \mathsf{Bool})\end{aligned}$$

Similar to a method, a field can be either a static field or an instance field as indicated by *isStatic*.

A field is uniquely identified by its qualified name.

### 2.2.10 Instructions

The full instruction set of Carmel is shown in Figure 2.3. The instruction set can be divided into six natural categories, called *fragments*, each handling a different aspect of Carmel: imperative core, objects, methods, arrays, subroutines, and exceptions.

Note that similar to JCVML most of the instructions are typed, i.e., they indicate the type of values expected and produced by the instruction. These types are called *operand types* and are defined as follows:

$$\mathsf{OpType} = \{\mathtt{r}, \mathtt{b}, \mathtt{s}, \mathtt{i}\}$$

representing references (or return addresses), `byte` (or `bool`) values, `short` values, and `int` values respectively.

In the figure square brackets are used to indicate an optional keyword or type declaration, e.g., `return` $[t]$ indicates that a `return` instruction may optionally return a value of type $t$.

### 2.2.11 Concrete Syntax

The abstract program structure defined in the preceding section is very convenient when defining the semantics for Carmel and other similarly formal tasks. However for illustration and concrete program examples it is less than optimal and does not convey the actual structure of a program in a very visual or intuitive way. Therefore a more readable concrete syntax (based on the syntax for Java Card) is used for program examples as illustrated in Figure 2.4.

For more details on the syntax, including a formal definition of the syntax as an extension of the Java syntax, see [Mar01, Section 3.3].

## 2.3 Bytecode Verification

One feature in particular makes Java Card, and indeed the entire family of Java-like languages, particularly attractive for developers of safety and security critical systems namely *bytecode verification*. The bytecode verification can be seen as an extended type check of JCVML code that in addition to ensuring "traditional" type safety also checks certain other safety properties, e.g., that programs are structurally well-formed and follows the type discipline of the virtual machine. This is very useful for making certain that compiled code that comes from untrusted or unknown sources can be checked independently to confirm that it is well-typed and structurally well-formed, i.e., that it is indeed a program. Furthermore the bytecode verification ensures that the verified program meets a number of requirements on the control and data flow of the program, e.g., that a local variable is not accessed before it has been assigned a

| | | | |
|---|---|---|---|
| | | *Imperative core* | |
| Instruction | ::= | `nop` | No operation |
| | \| | `push` $t$ $c$ | Push $c$ onto stack |
| | \| | `pop` $n$ | Pop $n$ stack positions |
| | \| | `dup` $m$ $n$ | Duplicate stack positions |
| | \| | `swap` $m$ $n$ | Swap stack positions |
| | \| | `numop` $t$ $op$ $[t']$ | Numerical operation |
| | \| | `goto` $pc$ | Unconditional jump to $pc$ |
| | \| | `if` $t$ $cmp$ $[nul]$ `goto` $pc$ | Conditional jump |
| | \| | `lookupswitch` $t$ $(k_i$`=>`$pc_i)_1^n$ `default=>`$pc_0$ | |
| | \| | `tableswitch` $t$ $l$`=>`$(pc_i)_0^n$ `default=>`$pc_{n+1}$ | |
| | \| | `load` $t$ $x$ | Load local variable |
| | \| | `store` $t$ $x$ | Save local variable |
| | \| | `inc` $t$ $x$ $c$ | Increment local variable |
| | | *Object fragment* | |
| | \| | `new` $\sigma$ | Create new instance |
| | \| | `checkcast` $t$ | Dynamic type-check |
| | \| | `instanceof` $\sigma$ | Dynamic type-check |
| | \| | `getstatic` $f$ | Fetch value of static field |
| | \| | `putstatic` $f$ | Update static field |
| | \| | `getfield` [`this`] $f$ | Fetch value of instance field |
| | \| | `putfield` [`this`] $f$ | Update instance field |
| | | *Method fragment* | |
| | \| | `invokestatic` $m$ | Invoke a static method |
| | \| | `invokevirtual` $m$ | Invoke a virtual method |
| | \| | `invokeinterface` *iface* | Invoke an interface |
| | \| | `return` $[t]$ | Return from invocation |
| | | *Arrays* | |
| | \| | `new (array` $t$`)` | Create new array |
| | \| | `arraylength` | Get length of array |
| | \| | `arrayload` $t$ | Fetch value from array |
| | \| | `arraystore` $t$ | Update array |
| | | *Subroutines* | |
| | \| | `jsr` $pc$ | Jump to subroutine |
| | \| | `ret` $x$ | Return from subroutine |
| | | *Exceptions* | |
| | \| | `throw` | Throw an exception |

Figure 2.3: Carmel instructions grouped by language fragment.

```
package p;

class Num
{
  short val;
}

class C
{
  static short rescue;

  short abs(Num n) {
    0: load r 1
    1: getfield Num.val
    2: store s 2
    3: goto 7
    4: pop 1
    5: getfield this p.C.rescue
    6: return s
    7: load s 2
    8: if lt s 0 goto 11
    9: load s 2
   10: return s
   11: load s 2
   12: numop s neg
   13: return s
   0-2: NullPointerExc => 4
  }
}

class C2 extends C
{
  short run() {
    0: push s 42
    1: putstatic C.rescue
    2: load r 0
    3: push s 3
    4: new (array p.Num)
    5: push s 1
    6: arrayload r
    7: invokevirtual p.C2.abs(p.Num)
    8: return s
  }
}
```

Figure 2.4: Example Carmel program

value and that the operand stack does not grow indefinitely. Following [HM01] the main tasks of the bytecode verifier can be summarised as follows:

- Stack frames do not under- of overflow
- Every bytecode is valid
- Jumps lead to legal instructions
- Method signatures contain valid information
- Operands are of the correct type
- Access control is obeyed
- Objects are initialised before use
- Subroutines are used in a first-in-first-out order

In addition, since the bytecode verifier is defined and implemented as a work-list based data flow analysis, it also ensures that when control flow paths are joined, e.g., after a conditional, the stack heights along the two paths must be the same and the local heaps must be type-compatible.

Since all JCVML programs must be verified before they are executed it is reasonable to also assume that all Carmel programs considered here have been bytecode verified. While this assumption makes some of the proofs in Section 5.2 much simpler by ensuring that the operand stack height for a given instruction is fixed, it is not strictly necessary since equivalent guarantees could be integrated in the analysis. Furthermore Carmel programs are well-formed by assumption and no other analysis nor the semantics defined in this chapter depends on programs being bytecode verified. Indeed some of the analyses could be used to perform many of the tasks performed by the bytecode verifier. However, this is tangential to the purpose of the present work and will not be discussed further here. The full specification and further details of the bytecode verifier can be found in [Sun00] and [LY99].

## 2.4 Operational Semantics

The formal semantics of the Carmel language can now be defined. The semantics is based on that of [Siv04, SH01] but modified to better suit the goals of this dissertation and clarify presentation as discussed in Section 2.1.

The Carmel semantics is specified as a structural operational semantics, i.e., a small step semantics, as described in [Plo81, NN92]. The presentation is structured around the six fragments of Carmel: imperative core, objects, methods, arrays, subroutines, and exceptions.

### 2.4.1 Semantic Domains

Values in Carmel programs are either numbers, references, or return addresses from subroutine jumps:

$$\mathsf{Val} = \mathsf{Num} + \mathsf{Ref} + \mathsf{RetAdr}$$

Since modelling the details of the different types of JCVML numbers, e.g., `byte` and `short`, is of no interest Carmel numbers are simply defined to be integers:

$$\mathsf{Num} = \mathbb{Z}$$

Subroutine jumps are always local to a method and therefore program counters suffice to implement return addresses:

$$\mathsf{RetAdr} = \mathsf{PC}$$

Program counters are taken to be the natural numbers and zero and it is assumed that programs are normalised in the sense that the program counter starts at zero for every method and that the instruction following program counter $pc$ can be found at $pc + 1$:

$$\mathsf{PC} = \mathbb{N}_0$$

For later use the domain of *addresses* is defined. An address is used to uniquely identify an instruction in a given program:

$$\mathsf{Addr} = \mathsf{Method} \times \mathsf{PC}$$

A reference is either a location in the heap or the `null`-reference:

$$\mathsf{Ref} = \mathsf{Location} \cup \{\texttt{null}\}$$

Locations are not specified further. It is assumed that the set of locations is countably infinite.

The heap contains objects and arrays during program execution and is implemented as a map from references to objects and arrays:

$$\mathsf{Heap} = \mathsf{Ref} \to (\mathsf{Object} + \mathsf{Array})$$

Objects are defined by the class they are instantiated from and a valuation of its instance fields:

$$\begin{aligned}\mathsf{Object} = {} & (\mathit{class} : \mathsf{Class}) \times \\ & (\mathit{fieldValue} : \mathsf{Field} \to \mathsf{Val})\end{aligned}$$

in the interest of succinctness and legibility the notation $o.f$ is used for an object, $o$, and a field $f \in \mathrm{dom}(o.\mathit{fieldValue})$, as a shorthand for $o.\mathit{fieldValue}(f)$ where no confusion can arise.

In JCVML arrays are actually considered as a special kind of objects. This is most clearly evidenced by the fact that JCVML allows method invocations

on array references in the same way as for object references. To improve clarity Carmel differentiates more clearly between arrays and objects.

An array contains a number of elements of a given type. The individual elements can be accessed through the *value* function:

$$\begin{aligned}\mathsf{Array} = (&\mathit{type} : \mathsf{ElemType})\times\\ (&\mathit{length} : \mathbb{N}_0)\times\\ (&\mathit{value} : \mathbb{N}_0 \to \mathsf{Val})\end{aligned}$$

Static fields are kept in a separate semantic component, called the static heap. Since static fields are uniquely identified by their name, type, and the class in which they are defined the static heap can be implemented directly as a map from fields to values:

$$\mathsf{StaHeap} = \mathsf{Field} \to \mathsf{Val}$$

The heap and the static heap can be seen as implementing the global variables in the sense that any object, array, or value stored in either the heap or the static heap is available to any method in the program with the right reference or field. In addition to these global variables, each method has a number of local variables that are stored in the local heap. The local variables do not have names as such but are identified simply by a number:

$$\mathsf{LocHeap} = \mathbb{N}_0 \to \mathsf{Val}$$

Note that the local variable with number 0 usually contains a reference to the object in which the method is invoked, i.e., a self-reference (sometimes called a `this`-pointer).

The notation $v_0 :: \cdots :: v_n$ is used as a shorthand for the local heap implemented by the map $[0 \mapsto v_0, \ldots, n \mapsto v_n]$.

Every method also has a local operand stack that is used as "scratch" space or for temporary storage. The stack is implemented as a sequence of values:

$$\mathsf{Stack} = \mathsf{Val}^*$$

By definition the left-most element of the sequence is the top element of the stack and thus the stack "grows" from right to left. The length of a stack $S = (v_0 :: \cdots :: v_i :: \cdots :: v_n)$ is written as $|S| = n + 1$ and the individual elements of a stack are accessed as $S|_i = v_i$ for $0 \leq i \leq n$.

Putting the above together the domain for stack frames can now be defined to consist of the method that is currently executing, the current program counter, a local heap, and an operand stack:

$$\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocHeap} \times \mathsf{Stack}$$

Exceptions that are not handled in the same method as it is thrown give rise to a special kind of stack frame, called an *exception frame*, that can only occur in the top-most position of a call stack. An exception frame only contains a

location pointing to an object of the exception class and the address where it was thrown (or re-thrown):

$$\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Addr}$$

For readability ordinary frames are written as $\langle m, pc, L, S\rangle$ and exception frames as $\langle \mathsf{Exc}\ loc_X, (m, pc)\rangle$. This then gives rise to the following definition of call stacks:

$$\mathsf{CallStack} = (\mathsf{Frame} + \mathsf{ExcFrame}) \times \mathsf{Frame}^*$$

which is simply a sequence of ordinary stack frames with either an ordinary or an exception frame as a top element. The notational conveniences used for operand stacks will also be used for call stacks.

Finally the domain for the semantic configurations of the operational semantics can be defined:

$$\mathsf{Conf} = \mathsf{RunConf} + \mathsf{FinConf}$$

Two different kinds of configurations are used in the semantics. The first kind is used for the normal execution of the semantics:

$$\mathsf{RunConf} = \mathsf{StaHeap} \times \mathsf{Heap} \times \mathsf{CallStack}$$

The second kind of configuration is used if and when a program terminates:

$$\mathsf{FinConf} = \mathsf{StaHeap} \times \mathsf{Heap} \times \mathsf{Val}_\perp$$

The value that replaces the call stack is used for a final return value or bottom if no value is returned.

Using the above definition for configurations the semantics of Carmel is defined as a structural operational semantics, cf. [Plo81]. For a program $P \in \mathsf{Program}$ the semantic reduction rules for are of the general form

$$P \vdash C \Longrightarrow C'$$

where $C, C' \in \mathsf{Conf}$. The notation $\Longrightarrow^*$ is used to indicate the reflexive and transitive closure of $\Longrightarrow$.

### 2.4.2 Programs and Initial Configurations

Before going into the details of the semantics of the Carmel instruction set, the initial configurations for Carmel programs must be defined. Since Java smart cards and JCVML allows for several applets to co-exist and run independently, a lengthy procedure is specified for downloading, installing, and registering an applet on a smart card. This is further complicated by the fact that a given applet may be instantiated more than once on a given smart card. For these reasons JCVML does not have a single special method that acts as a starting point for all programs. Instead each applet defines a method that is designated as the applets *entry point* to be invoked when the applet is executed.

A more abstract approach is taken here: every package is considered to be an applet and applets can only be instantiated once. Furthermore every package must define a special class, called the *main* class, that implements a method designated as the *entry point* of that package. Execution of a program then starts by instantiating all of the special classes and invoking one of the entry points with a self-reference.

For a package $p \in P.packages$ in a given program, $P \in \mathsf{Program}$, the main class is accessed by the $main : \mathsf{Package} \to \mathsf{Class}$ function and let $P.main$ denote the set of all main classes in a program. The entry point of a class, $\sigma \in \mathsf{Class}$, is similarly accessed by the function $entry : \mathsf{Class} \to \mathsf{Method}$.

The initial configurations for Carmel programs can then be formalised as follows:

**Definition 2.1 (Initial configurations).** *For a program, $P \in \mathsf{Program}$, the configuration $C \in \mathsf{RunConf}$ is an* initial configuration *if and only if $C$ can be written on the form $\langle K, H, \langle m_\sigma, 0, [0 \mapsto loc_\sigma], \epsilon\rangle :: \epsilon\rangle$ where $\sigma \in P.main$, $m_\sigma = \sigma.entry$, and $\forall \tau \in P.main : \exists loc_\tau : H(loc_\tau) = \tau$.*

Note that a program can have several different initial configurations, each corresponding to a different applet being activated.

### 2.4.3 Imperative Core

The imperative core of Carmel is concerned with manipulating the stack and local heap and contains the basic control structures such as conditionals. Figures 2.5 and 2.6 display the full imperative core semantics.

The `nop` instruction is a "no-op" or "dummy" instruction that does not affect the result of a program. It is included for completeness. For stack manipulation the usual `push` and `pop` instructions are provided. The instructions `dup` and `swap` implement additional instructions for changing the stack: `dup` duplicates the top $m$ words at stack position $n$ and `swap` interchanges the top $m$ elements with the immediately following $n$ elements on the stack.

Arithmetic is performed by the `numop` instruction that takes one or two elements from the top of the stack, depending on whether the operation to be performed is unary or binary, and replaces the element(s) with the result of performing the numerical operation. The binary operators supported are $\mathsf{BinaryOp} = \{\texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{div}, \texttt{rem}, \texttt{cmp}, \texttt{and}, \texttt{or}, \texttt{xor}, \texttt{shl}, \texttt{shr}\}$ and the unary operators $\mathsf{UnaryOp} = \{\texttt{neg}, \texttt{to}\}$. Note that division by zero is explicitly disallowed and will result in a stuck configuration. In a later section runtime exceptions are discussed which, among other things, allows an exception to be raised when a division by zero is detected instead of ending in a stuck configuration.

Carmel has several instructions for control flow: `goto` for unconditional jumps, `if` for conditional jumps, and `lookupswitch` and `tableswitch` for case analysis. Conditionals come in two flavours: one that compares the top two elements of the stack and one that compares the top element to either the null reference (`null`) or the number zero (0) depending on the type. Note that when comparing references only the operators `eq` for equality and `ne` for inequality

$$\frac{m.instructionAt(pc) = \texttt{nop}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ t\ \ c}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, c :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{pop}\ \ n}{\begin{array}{r}P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle m, pc+1, L, S\rangle :: SF\rangle\end{array}}$$

$$\frac{\begin{array}{c}m.instructionAt(pc) = \texttt{dup}\ \ m\ \ n \\ m > 0 \qquad |S_1| = m \qquad |S_1 :: S_2| = n\end{array}}{\begin{array}{r}P \vdash \langle K, H, \langle m, pc, L, S_1 :: S_2 :: S_3\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle m, pc+1, L, S_1 :: S_2 :: S_1 :: S_3\rangle :: SF\rangle\end{array}}$$

$$\frac{\begin{array}{c}m.instructionAt(pc) = \texttt{swap}\ \ m\ \ n \\ m > 0 \qquad n > 0 \qquad |S_1| = m \qquad |S_2| = n\end{array}}{\begin{array}{r}P \vdash \langle K, H, \langle m, pc, L, S_1 :: S_2 :: S_3\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle m, pc+1, L, S_2 :: S_1 :: S_3\rangle :: SF\rangle\end{array}}$$

$$\frac{\begin{array}{c}m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{BinaryOp} \qquad v = op(v_1, v_2) \qquad op \in \{\texttt{div}, \texttt{rem}\} \Rightarrow v_2 \neq 0\end{array}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle K, S, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c}m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{UnaryOp} \qquad v = op(v_1)\end{array}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: S\rangle :: SF\rangle \Longrightarrow \langle K, S, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

Figure 2.5: Imperative core semantics (1)

$$\frac{m.instructionAt(pc) = \texttt{goto}\ \ pc_0}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc_0, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0 \qquad t = \texttt{r} \Rightarrow cmp \in \{\texttt{eq}, \texttt{ne}\} \qquad pc_1 = \begin{cases} pc_0 & \text{if } cmp(v_1, v_2) = true \\ pc + 1 & \text{otherwise} \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc_1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmp\ \ nul\ \ \texttt{goto}\ \ pc_0 \qquad t = \texttt{r} \Rightarrow cmp \in \{\texttt{eq}, \texttt{ne}\} \qquad pc' = \begin{cases} pc_0 & \text{if } cmp(v_1, nul) = true \\ pc + 1 & \text{otherwise} \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc', L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{lookupswitch}\ \ t\ \ (k_i\texttt{=>}pc_i)_1^n\ \ \texttt{default=>}pc_0 \qquad pc' = \begin{cases} pc_i & \text{if } \exists i \in \{1, \ldots, n\} : v = k_i \\ pc_0 & \text{otherwise} \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc', L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{tableswitch}\ \ t\ \ l\texttt{=>}(pc_i)_0^n\ \ \texttt{default=>}pc_{n+1} \qquad pc' = \begin{cases} pc_i & \text{if } l \leq v \leq (l + n) \text{ and } i = v - l \\ pc_{n+1} & \text{otherwise} \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc', L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{load}\ \ t\ \ x}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc + 1, L, L(x) :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{store}\ \ t\ \ x}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc + 1, L[x \mapsto v], S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{inc}\ \ t\ \ x\ \ c}{\begin{array}{l} P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle K, H, \langle m, pc + 1, L[x \mapsto L(x) + c], S\rangle :: SF\rangle \end{array}}$$

Figure 2.6: Imperative core semantics (2)

are allowed. The `lookupswitch` compares the top element of the stack with a list of keys and jumps to the corresponding address if a match is found, otherwise it jumps to the specified default address. The second switching instruction, `tableswitch`, first subtracts a base value from the top element of the stack and then uses the result as an index into a jump table.

The `load` instruction is provided for reading the content of a local variable and push it onto the top of the operand stack. Conversely the `store` instruction updates the value of the specified local variable with the value on top of the stack. Finally the `inc` instruction is a convenient shortcut for performing addition and subtraction directly on values stored in local variables without the need to go through the operand stack.

### 2.4.4 Object Fragment

The object fragment contains instructions for creating objects, runtime type-checks, and field manipulation (both static and instance). The semantics of the object fragment is shown in Figure 2.7.

The `new` instruction is used to create new instances of the class taken as a parameter. The instruction allocates an unused location on the heap and returns it on top of the stack. This is formalised in the *newObject* function:

$$\begin{aligned} &\mathit{newObject} : \mathsf{Class} \times \mathsf{Heap} \rightarrow \mathsf{Location} \times \mathsf{Heap} \\ &\mathit{newObject}(\sigma, H) = (\mathit{loc}, H') \end{aligned}$$

where

$$\mathit{loc} \notin \mathrm{dom}(H) \quad \wedge \quad o \in \mathsf{Object} \quad \wedge \quad H' = H[\mathit{loc} \mapsto o] \quad \wedge \quad o.\mathit{class} = \sigma$$

The instance fields of the new object must be initiated with the correct default value:

$$\forall f \in \sigma.\mathit{fields} : \neg f.\mathit{isStatic} \Rightarrow o.\mathit{fieldValue}(f) = \mathit{def}(f.\mathit{type})$$

The default value of a field depends on its type:

$$\mathit{def}(t) = \begin{cases} 0 & \text{if } t \in \mathsf{PrimType} \\ \texttt{null} & \text{if } t \in \mathsf{RefType} \end{cases}$$

Due to the dynamic nature of object creation it is sometimes useful to be able to check the type of an object at runtime. Carmel provides two instructions for this: `checkcast` and `instanceof`. While both instructions basically check the same thing, they handle failure very differently. The `checkcast` instruction will become stuck if the type-check fails, either because $\mathit{loc} = \texttt{null}$ or because the object pointed to does not belong to a subclass of $\sigma$; in a later section exceptions are added to the semantics allowing this instruction to throw an exception instead of becoming stuck. In contrast `instanceof` does not get stuck upon failure but simply return a 0 on top of the stack and 1 if the check was successful.

$$\frac{m.instructionAt(pc) = \texttt{new}\ \sigma \qquad \sigma \in \mathsf{Class} \qquad (loc, H') = newObject(\sigma, H)}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H', \langle m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{checkcast}\ \sigma \qquad loc \neq \texttt{null} \Rightarrow H(loc).class \preceq \sigma}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{instanceof}\ \sigma \\ v = \begin{cases} 1 & \text{if } loc \neq \texttt{null} \wedge H(loc).class \preceq \sigma \\ 0 & otherwise \end{cases} \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getstatic}\ f \\ f.isStatic = true \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, S, \langle m, pc+1, L, K(f) :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putstatic}\ f \\ f.isStatic = true \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle K[f \mapsto v], H, \langle m, pc+1, L, S\rangle :: S\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield}\ f \\ loc \neq \texttt{null} \qquad o = H(loc) \qquad v = o.fieldValue(f) \qquad f.isStatic = false \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield}\ f \\ f.isStatic = false \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ o' = o[fieldValue \mapsto o.fieldValue[f \mapsto v]] \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle m, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle K, H[loc \mapsto o'], \langle m, pc+1, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield this}\ f \\ f.isStatic = false \qquad loc = L(0) \qquad loc \neq \texttt{null} \\ o = H(loc) \qquad v = o.fieldValue(f) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield this}\ f \\ f.isStatic = false \qquad loc = L(0) \qquad loc \neq \texttt{null} \\ o = H(loc) \qquad o' = o[fieldValue \mapsto o.fieldValue[f \mapsto v]] \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle K, H[loc \mapsto o'], \langle m, pc+1, L, S\rangle :: SF\rangle}$$

Figure 2.7: Object fragment semantics

Two instructions are provided for manipulating static fields: `getstatic` and `putstatic`. The `getstatic` instruction reads the contents of a static field and pushes it onto the stack and vice versa for `putstatic`.

The instructions for reading and writing instance fields mirror those for statics fields. The two instructions, `getfield` and `putfield`, are used to read and update instance fields respectively. Because instance fields are local to the object that contains them an additional object reference (location) is needed to identify the correct instance field. Since methods often need to reference instance fields in the current object, i.e., the same object as the one the method is invoked in, two variants of the instructions for working with instance fields are provided: `getfield this` and `putfield this`. These instructions use the `self` reference found in local variable 0 as object reference to locating the specified field rather than taking it as an explicit parameter on the operand stack.

### 2.4.5 Method Fragment

The method fragment contains all instructions concerning method invocation and return, see Figure 2.8.

As discussed in Section 2.2.7, Carmel supports two kinds of methods: static and virtual invoked by `invokestatic` and `invokevirtual` respectively.

Since static methods are resolved at compile time, no method lookup is needed in order to find the right method. This is different for virtual methods: in order to identify the virtual method to be invoked, a reference (location) to the relevant object must be provided. The corresponding method is then looked up in the class hierarchy and executed, this is called *dynamic dispatch* since resolution is done dynamically at runtime. Method lookup is implemented by the following function:

$$
\begin{aligned}
&\mathit{methodLookup}(m_0, \sigma) = \\
&\quad \begin{cases}
\bot & \text{if } \sigma = \bot \\
m_0 & \text{if } m_0 \in \sigma.\mathit{methods} \wedge \sigma \neq \bot \\
\mathit{methodLookup}(m_0, \sigma.\mathit{super}) & \text{if } m_0 \notin \sigma.\mathit{methods} \wedge \sigma \neq \bot
\end{cases}
\end{aligned}
$$

Conceptually interfaces are a way to reclaim some of the power of multiple inheritance, by allowing a class to implement several interfaces from which the class then "inherits" methods and fields, and yet avoiding the problems and ambiguities inherent in multiple inheritance.

When a virtual method (or an interface) returns, the stack frame of the invoking (virtual) method is reinstated with the arguments and object reference for the invoked method removed from the operand stack and control is returned to the invoking method. In [Siv04, SH01] the arguments and object reference are removed from the stack immediately when the method is invoked rather than when it returns. The choice to leave the arguments and reference on the operand stack until the method returned was made in order to simplify the proof of correctness for the control flow analysis specified in Chapter 3. Note that this does not in any way change the behaviour of Carmel programs it is

$$\frac{m.instructionAt(pc) = \texttt{invokestatic}\ m_0 \quad m.isStatic = true \quad n = |m_0| \quad L_0 = v_1 :: \cdots :: v_n}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m_0, 0, L_0, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{invokevirtual}\ m_0 \quad m.isStatic = false \quad loc \neq \texttt{null} \quad o = H(loc) \quad L_v = loc :: v_1 \cdots :: v_{|m_0|} m_v = methodLookup(m_0, o.class)}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m_v, 0, L_v, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{invokeinterface}\ m_0 \quad loc \neq \texttt{null} \quad o = H(loc) \quad L_v = loc :: v_1 \cdots :: v_{|m_0|} \quad m_v = methodLookup(m_0, o.class)}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m_v, 0, L_v, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return} \quad S' = \begin{cases} v'_1 :: \cdots :: v'_{|m|} :: loc :: S'' & \text{if } m.isStatic \neq true \\ v'_1 :: \cdots :: v'_{|m|} :: S'' & \text{if } m.isStatic = true \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m', pc' + 1, L', S''\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return}\ t \quad S' = \begin{cases} v'_1 :: \cdots :: v'_{|m|} :: loc :: S'' & \text{if } m.isStatic \neq true \\ v'_1 :: \cdots :: v'_{|m|} :: S'' & \text{if } m.isStatic = true \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m', pc' + 1, L', v :: S''\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: \epsilon\rangle \Longrightarrow \langle K, H, \langle \mathsf{Ret}\ \bot\rangle\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{return}\ t}{P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: \epsilon\rangle \Longrightarrow \langle K, H, \langle \mathsf{Ret}\ v\rangle\rangle}$$

Figure 2.8: Method fragment semantics

$$\frac{m.instructionAt(pc) = \texttt{new (array } t\texttt{)} \quad n \geq 0 \quad (H', loc) = newArray(t, n, H)}{P \vdash \langle K, H, \langle m, pc, L, n :: S\rangle :: SF\rangle \Longrightarrow \langle K, H', \langle m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arraylength} \quad loc \neq \texttt{null} \quad n = H(loc).length}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, n :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arrayload } t \quad loc \neq \texttt{null} \quad v = H(loc).value(n) \quad 0 \leq n < H(loc).length}{P \vdash \langle K, H, \langle m, pc, L, n :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arraystore } t \quad loc \neq \texttt{null} \quad a = H(loc) \quad a' = a[value \mapsto a.value[n \mapsto v]] \quad H' = H[loc \mapsto a'] \quad 0 \leq n < H(loc).length}{P \vdash \langle K, H, \langle m, pc, L, v :: n :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H', \langle k, pc+1, L, S\rangle :: SF\rangle}$$

Figure 2.9: Array fragment semantics

merely an annotation. However, as a consequence the "clean up" when returning from a method invocation is different for virtual and static methods. For static methods the parameters on the stack must be removed, for virtual methods the object reference must be removed as well. In addition there are two variants of the `return` instruction: one for methods that do not return a value on for methods that do.

Finally, the `return` instructions must also handle the special case of program termination, i.e., when the returning method is actually the method invoked in the initial configuration. In that case the `return` instructions evaluate to a final configuration.

### 2.4.6 Arrays

The support for arrays in Carmel, and in JCVML, is rather rudimentary and is limited to creating arrays, determining the length of an array, and reading and modifying array elements, see Figure 2.9.

Creating a new array is done using the `new (array` $t$`)` in a manner that is very similar to the creation of a new object using the *newArray* function:

$$newArray : \mathsf{Type} \times \mathbb{N} \times \mathsf{Heap} \rightarrow \mathsf{Location} \times \mathsf{Heap}$$
$$newArray(t, n, H) = (loc, H')$$

where

$$loc \notin \mathrm{dom}(H) \ \wedge \ a \in \mathsf{Array} \ \wedge \ H' = H[loc \mapsto a] \ \wedge \ a.type = t \ \wedge \ a.length = n$$

$$\frac{m.instructionAt(pc) = \texttt{jsr}\ \ pc_0}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc_0, pc+1 :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{ret}\ \ n \qquad pc_0 = L(n)}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc_0, V, S\rangle :: SF\rangle}$$

Figure 2.10: Subroutine semantics

Furthermore, the array is initialised according to its type: $\forall i \in \{0, \ldots, n-1\} : a.value(i) = def(t)$.

The length of an array is readily obtained using the `arraylength` instruction and elements are read and written using the `arrayload` and `arraystore` instructions respectively.

### 2.4.7 Subroutines

Subroutines are not, as such, formally defined entities, e.g., like methods, in Carmel nor in JCVML. Instead subroutines are implemented by a special jump instruction, `jsr`, that stores the jump address on the stack (actually the address immediately following the jump instruction). It is then the responsibility of the target of the jump to store the would be return address and, when done, use the return instruction, `jsr`, to jump back to the return address.

See Figure 2.10 for the semantics.

### 2.4.8 Exceptions

Exceptions can arise in two different ways in JCVML: a programmer may explicitly throw an exception or an exception may be thrown by the runtime system as a consequence of some error condition. For the developments in subsequent chapters only few very specific runtime exceptions, if any, are of interest. For this reason, and in the interest of keeping the language small, runtime exceptions are not considered to be part of Carmel proper. Instead a Carmel program will end in a stuck configuration if an unrecoverable runtime error occurs. However, for completeness sake Section 2.5 shows how runtime exceptions can be added to the basic Carmel semantics. The resulting language is called $\text{Carmel}_{\mathsf{EXC}}$. Section 3.5 illustrates how a control flow analysis for Carmel can be extended to an exception analysis handling both programmer exceptions and runtime exceptions and thus handle all of $\text{Carmel}_{\mathsf{EXC}}$.

Exception handlers, like methods, are defined directly in the program structure and thus, there are no instructions, as such, for defining exception handlers just as there are no specific instructions for defining a method. Therefore there is only one instruction relating to exceptions, namely the `throw` instruction.

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{throw} \\ loc \neq \texttt{null} \qquad \sigma_X = H(loc).class \preceq \texttt{Throwable} \\ F = \begin{cases} \langle m, pc_0, L, loc :: \epsilon \rangle & \text{if } findHandler(m, pc, \sigma_X) = pc_0 \\ \langle \mathsf{Exc}\ loc, (m, pc) \rangle & \text{if } findHandler(m, pc, \sigma_X) = \bot \end{cases} \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S \rangle :: SF \rangle \Longrightarrow \langle K, H, F :: SF \rangle}$$

$$\frac{\sigma_X = H(loc_X).class \qquad findHandler(m, pc, \sigma_X) = pc_0}{\begin{array}{r} P \vdash \langle K, H, \langle \mathsf{Exc}\ loc, (m_X, pc_X) \rangle :: \langle m, pc, L, S \rangle :: SF \rangle \Longrightarrow \\ \langle K, H, \langle m, pc_0, L, loc_X :: \epsilon \rangle :: SF \rangle \end{array}}$$

$$\frac{\sigma_X = H(loc_X).class \qquad findHandler(m, pc, \sigma_X) = \bot}{\begin{array}{r} P \vdash \langle K, H, \langle \mathsf{Exc}\ loc_X, (m_X, pc_X) \rangle :: \langle m, pc, L, S \rangle :: SF \rangle \Longrightarrow \\ \langle K, H, \langle \mathsf{Exc}\ loc_X, (m, pc) \rangle :: SF \rangle \end{array}}$$

Figure 2.11: Exception semantics

Exceptions are implemented simply as objects that belong to a subclass of the `Throwable` class. The `throw` instruction either jumps to a local exception handler if one is found or, if the exception can not be handled locally, replaces the current stack frame either with a special *exception frame* that contains only the location pointing to the exception object, cf. Figure 2.11.

Two rules that are not directly related to instructions are also needed to evaluate semantic configurations with an exception frame on top of the call stack: one that applies when an exception handler was found in the method of the stack frame below the exception frame and one that applies when no exception handler was found. Note that the latter case conceptually corresponds to re-throwing the exception and this is noted by updating the address in the exception frame.

In order to define the *findHandler* used above to find the appropriate local handler, if one exists, two auxiliary predicates are defined. The first determines if a given handler, $H$, can handle an exception of class $\sigma_X$ that was thrown at program counter *pc*:

$$\begin{array}{rl} canHandle(h, pc, \sigma_X) \equiv & h.startAddr \leq pc \leq h.endAddr \wedge \\ & (\sigma_X \preceq h.catchType \vee h.catchType = \bot) \end{array}$$

The second predicate checks if a given handler is the first applicable handler:

$$\begin{array}{rl} firstHandler(H, i, pc, \sigma_X) \equiv & canHandle(H(i), pc, \sigma_X) \wedge \\ & (\forall j : canHandle(H(j), pc, \sigma_X) \ \Rightarrow \ j \leq i) \end{array}$$

Given the above two predicates the *findHandler* function is defined as follows:

$$findHandler(m, pc, \sigma_X) = \begin{cases} H(i).handlerAddr & \text{if } H = m.handlers \wedge \\ & \quad \exists i : firstHandler(H, i, pc, \sigma_X) \\ \bot & \text{otherwise} \end{cases}$$

## 2.5 Carmel$_{\mathsf{EXC}}$ and Runtime Exceptions

In contrast to exceptions explicitly thrown, using the `throw` instruction, by request of the programmer, runtime exceptions are thrown implicitly by the runtime system when an error is detected, e.g., if a program tries to divide a number by 0 or dereference a `null` pointer. Many of the semantic rules described in the previous section included in their premise extra conditions to prevent such errors from occurring resulting instead in a stuck configuration. In this section we introduce *runtime exceptions* to handle the cases where the ordinary semantics would get stuck by automatically throwing an exception. This extension of the Carmel semantics is called Carmel$_{\mathsf{EXC}}$.

### 2.5.1 Preallocated Exceptions

Runtime exceptions are handled by the same semantic structures and rules as ordinary exceptions. But since runtime exceptions occur implicitly it is not possible to directly copy the approach taken for ordinary exceptions that require programs to explicitly allocate an exception object before throwing it. Instead it is assumed that the runtime system has preallocated exception objects for the runtime exception classes.

The set of runtime exceptions that are assumed to be preallocated is defined as follows:

$$\begin{aligned}\mathsf{RuntimeException} = \{&\texttt{ArithmeticExc},\\ &\texttt{NullPointerExc},\\ &\texttt{ClassCastExc},\\ &\texttt{NegArraySizeExc},\\ &\texttt{IndexOutOfBoundsExc}\}\end{aligned}$$

A reference to one of these preallocated objects can be obtained by using the *excLocation*:

$$\mathit{excLocation} : \mathsf{Heap} \times \mathsf{RuntimeException} \rightarrow \mathsf{Location}$$

In the next section the semantic rules are discussed for a few of the instructions that may throw a runtime exception. The remaining instructions can be found in Appendix A.1.

### 2.5.2 Extending the Semantics

Below a few examples are given to show how the semantic rules can be extended to throw the proper runtime exception when an error is detected.

To simplify the specification the *nextFrame* function is introduced:

$$\begin{aligned}&\mathit{nextFrame}(m, pc, L, S, H, \mathit{loc}_X, \sigma_X) =\\ &\quad\begin{cases}\langle m, pc_0, L, \mathit{loc}_X :: \epsilon\rangle & \text{if } \mathit{findHandler}(m, pc, \sigma_X) = pc_0\\ \langle \mathsf{Exc}\ \mathit{loc}_X, (m, pc)\rangle & \text{if } \mathit{findHandler}(m, pc, \sigma_X) = \bot\end{cases}\end{aligned}$$

Intuitively the *nextFrame* function jumps to a local exception handler if one exists for the given exception. Otherwise an exception frame is returned.

Only division by zero gives rise to a runtime exception from numerical operators, namely the `ArithmeticExc` exception. This is formalised in the following rule:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ t\ \ op\ \ [t'] \\ op \in \{\texttt{div}, \texttt{rem}\} \qquad v_2 = 0 \qquad \sigma_X = \texttt{ArithmeticExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

Most of the instructions that require an object reference as an argument will throw an exception, `NullPointerExc`, if the null reference is passed, including the `throw` instruction. This is exemplified below in the `getfield` instruction:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield}\ f \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

The `checkcast` instruction checks if an object belongs to a subclass of the class given as an argument. If this is not the case the instruction throws the `ClassCastExc` exception:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{checkcast}\ \sigma \\ loc \neq \texttt{null} \Rightarrow H(loc).class \not\preceq \sigma \qquad \sigma_X = \texttt{ClassCastExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

Thus a `checkcast` instruction can be seen as an assertion or a condition for the program to continue execution.

Arrays must have a non-negative size otherwise the `NegArraySizeExc` is thrown when an array is created:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{new (array}\ t\texttt{)} \\ n < 0 \qquad \sigma_X = \texttt{NegArraySizeExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, n :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

Finally, accessing array elements can fail in two ways: by passing a null reference, resulting in a `NullPointerExc` exception, or by accessing an index that is not within the bounds of the array, resulting in an `IndexOutOfBoundsExc` exception.

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{arrayload}\ t \\ \sigma_X = \begin{cases} \texttt{NullPointerExc} & \text{if } loc = \texttt{null} \\ \texttt{IndexOutOfBoundsExc} & \text{if } n < 0 \vee n \geq H(loc).length \end{cases} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, n :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

The remaining rules that may throw runtime exceptions can be found in Appendix A.1.

$$
\begin{array}{rcl}
\mathsf{Instruction}_{\mathsf{Core}} & ::= & \texttt{push}\ t\ c \\
& | & \texttt{pop}\ n \\
& | & \texttt{numop}\ t\ op\ [t'] \\
& | & \texttt{goto}\ pc \\
& | & \texttt{if}\ t\ cmp\ [nul]\ \texttt{goto}\ pc \\
& | & \texttt{load}\ t\ x \\
& | & \texttt{store}\ t\ x \\
& | & \texttt{new}\ \sigma \\
& | & \texttt{getfield}\ f \\
& | & \texttt{putfield}\ f \\
& | & \texttt{invokevirtual}\ m \\
& | & \texttt{return}\ [t]
\end{array}
$$

Figure 2.12: Carmel Core instructions

## 2.6 Carmel Core

Even though Carmel represents a significant rationalisation over JCVML, including a sizable reduction in the number of instructions from 100+ to just 31, it is still a rather large language for use with formal methods and in particular for manual proofs. For this reason a subset of Carmel, called Carmel Core, is identified that contains only essential structures and instructions. This core language, in essence a Carmel (or JCVML) calculus, is used in later chapters to develop and study advanced analyses and properties. The full instruction set of Carmel Core is shown in Figure 2.12. Note that Carmel Core is comparable to the Java/JVML subsets studied, e.g., in [FM99, FM98, RR98, WR99, IPW99, ZR02, CHS01].

The semantics for Carmel Core is a simplified version of the Carmel semantics. In particular the program structure and semantic domains for Carmel Core can be constructed as restrictions of the corresponding structures and domains for Carmel. For example, since exceptions are not supported no exception frames are needed. As another example neither static fields nor methods are included in Carmel Core and the static heap is therefore not included either. The reduction rules for Carmel Core can be found in Appendix A.2.

## 2.7 Summary

In this chapter the program structure, syntax, and semantics for the Carmel language was introduced. Carmel is a rational reconstruction of the Java Card Virtual Machine Language that will be used as the target language for the problems, properties, and analyses developed in this dissertation. A small subset of Carmel, called Carmel Core, was defined to facilitate and simplify later developments of advanced analyses and properties.

In [SJ03a] a prototype implementation of an interpreter for Carmel is discussed. The interpreter supports step-by-step visualisation of the execution of a program and makes it inspect all parts of the state of the virtual machine, e.g., the operand stack and the heap, during program execution. The prototype also implements the control flow analysis for Carmel specified in Section 3.2.

While the formalisation of the Java Virtual Machine has been a very active area of research, with [Ber98, Ber97, Ste98] as early examples, the formalisation of the Java Card Virtual Machine Language is less well-studied. A formal operational semantics for a precursor to JCVML, the Java Secure Processor, is given in [HBL99]. For a comprehensive survey of the considerable amount of research into various aspects of Java and related subjects see [HM01].

Chapter 3

# Flow Logic for Carmel

*Logic merely enables one to be wrong with authority*
*—Doctor Who*

This chapter serves as an introduction to the concept of *Flow Logic*, a specification oriented approach to static analysis. The Flow Logic framework is introduced by way of formally developing and proving correct a *control flow analysis* for Carmel in great detail. The analysis is furthermore used as a foundation for other analyses developed in subsequent chapters. The use of the control flow analysis as basis is exemplified in this chapter by extending it to an *exception analysis* covering the entire $\text{Carmel}_{\mathsf{EXC}}$ language.

In addition it is shown how to systematically turn the abstract Flow Logic specification into a *constraint generator* over the Alternation-free Least Fixed-Point logic for which solutions can be found efficiently. The constraint generator is formally proved to be equivalent to the Flow Logic specification and thus the formal correctness of the analysis is carried through to the implementation. Such an "end-to-end" guarantee of correctness is very important for applications that require a very high degree of formal assurance, e.g., high security systems evaluated according to the Common Criteria security evaluation standard [CC99], where the higher assurance levels (EAL5 and up) require verification and validation based on formal methods.

Formal verification and validation is often demanding and time-consuming work and consequently very expensive. In this dissertation it is argued that the framework and general methodology presented and discussed here can be leveraged to significantly reduce the cost of such verification and validation tasks by facilitating the construction of automated verification and validation tools. Here the basic tenet of static analysis, that an analysis ultimately must be implementable (or at least decidable), lends a unique perspective to the de-

velopment of such tools. Development is further supported by the the fact that trade-offs between cost and precision of an analysis are well-studied and well-understood; this is exemplified in the Galois connections often used in abstract interpretation, cf. [CC77, NNH99], that provide theory and tools for systematically constructing less precise (and thus less costly) analyses from more precise analyses and thereby allowing a systematic exploration of the design space for a given analysis.

## 3.1 Flow Logic

Flow Logic is a constraint-based specification oriented framework for developing static analyses [NN97, NN98c, NNH99, NN02]. The applicability of the framework is demonstrated by the wide variety of languages and calculi for which analyses have been specified, including the $\pi$-calculus [BDNN01b, BDNN99, BDNN98], Spi-calculus [NNS02a], $\lambda$-calculus [NN98b, NN97], the ambient calculus [NNH02, NNHJ99, NHN03, HJNN99], imperative objects [NN98a], Concurrent ML [GNN97], and protocol narrations [BBD$^+$03].

In the Flow Logic framework a clear distinction is made between *specifying* and analysis and *computing* an analysis result, not unlike type-systems. Flow Logic is, in general, used only for the former while the latter is a separate activity often involving other formalisms and tools. This approach allows efforts to be focused on designing and specifying an analysis without making compromises dictated by implementation considerations. On the other hand implementation of an analysis also benefits since Flow Logic is "implementation agnostic" in the sense that no particular tool or formalism is proscribed by the framework. The implementor of an analysis is thus free to choose a suitable tool and in particular is able to exploit emerging and state of the art technologies.

The flexibility afforded by the high-level logical approach also allows insights gained from other approaches, such as abstract interpretation and type/effect systems, to be integrated rather easily into Flow Logic specifications.

## 3.2 Control Flow Analysis

An important feature of many object-oriented languages, including Carmel, is that of *dynamic dispatch* as mentioned in Section 2.4.5. One of the immediate consequences of dynamic dispatch is that the actual control flow and call graph of a program is resolved dynamically at runtime. However, in order to apply the techniques from program analysis to verify and validate safety and security properties of programs it is a prerequisite that the control flow of a program is known (or at least a conservative approximation thereof). In particular: for every method (or interface) invocation in a program the set of methods that can actually be invoked at that point must be known statically. This leads to the notion of a *control flow analysis* which is a program analysis that for each method invocation in a program computes an over-approximation of the set of

methods that can be invoked at that point. In [TP00] a number of control flow analyses for object-oriented languages are described (re-formulated as set based analyses) and compared with respect to speed and precision.

In this section a control flow analysis is developed for Carmel. For presentation purposes the basic analysis is specified for Carmel without taking exceptions into account; Section 3.5 then extends the basic analysis to handle both user thrown and runtime exceptions. A general strategy for implementing analyses specified in the Flow Logic framework is also discussed and illustrated in detail for the control flow analysis.

### 3.2.1 Preliminaries

In anticipation of the following sections, the most important lattice and order theoretic notions used are reviewed below. Since the notions and definitions are standard they are not discussed in any detail. For a through treatment see [DP90, NNH99].

**Definition 3.1 (Partial order).** *A* partial order *in $P$ is a relation, $\sqsubseteq$, on $P$ such that $\sqsubseteq$ is reflexive, anti-symmetric, and transitive:*

1. $\forall x \in P\colon x \sqsubseteq x$
2. $\forall x, y \in P\colon x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3. $\forall x, y, z \in P\colon x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

*A* partially ordered set *is a set $P$ equipped with a partial order $\sqsubseteq$.*

If $P$ has an element $x \in P$ such that $\forall y \in P\colon x \sqsubseteq y$ then this element is called the *least element* of $P$ and is denoted $\bot$ (or even $\bot_P$). Analogously, the *greatest element* of $P$ is an element $x \in P$ such that $\forall y \in P\colon y \sqsubseteq x$ and is denoted $\top$ (or $\top_P$). Generalising this leads to the definition of upper bounds:

**Definition 3.2 (Upper bound).** *Let $(P, \sqsubseteq)$ be a partially ordered set and let $S \subseteq P$, then $u \in P$ is an* upper bound *for $S$ in $P$ if $\forall x \in S\colon x \sqsubseteq u$. If furthermore $u \sqsubseteq v$ for all upper bounds $v$ of $S$ in $P$ then $u$ is the* least upper bound *of $S$ in $P$ and is denoted $\bigsqcup S$ whenever it exists.*

The binary least upper bound $\bigsqcup\{x, y\}$ is written $x \sqcup y$. The converse notion of lower bounds can be defined similarly:

**Definition 3.3 (Lower bound).** *Let $(P, \sqsubseteq)$ be a partially ordered set and let $S \subseteq P$, then $l \in P$ is a* lower bound *for $S$ in $P$ if $\forall y \in S\colon l \sqsubseteq y$. If furthermore $m \sqsubseteq l$ for all lower bounds $m$ of $S$ in $P$ then $l$ is the* greatest lower bound *of $S$ in $P$ and is denoted $\bigsqcap S$ whenever it exists.*

The binary greatest lower bound $\bigsqcap\{x, y\}$ is written $x \sqcap y$.

**Definition 3.4 (Lattice).** *Let $(P, \sqsubseteq)$ be a partially ordered set such that $P \neq \emptyset$. If $x \sqcup y$ and $x \sqcap y$ exists for all $x, y \in P$, then $(P, \sqsubseteq)$ is a* lattice.

**Definition 3.5 (Complete lattice).** *Let $(P, \sqsubseteq)$ be a partially ordered set such that $P \neq \emptyset$. If $\bigsqcup S$ and $\bigsqcap S$ exists for all $S \subseteq P$, then $(P, \sqsubseteq)$ is a* complete lattice.

Note that if $P$ is a complete lattice then $\bot = \bigsqcup \emptyset = \bigsqcap P$ and $\top = \bigsqcap \emptyset = \bigsqcup P$.

### 3.2.2 Abstract Domains

An important part of designing an analysis is to find and define abstract domains that can support the kind of analysis being developed. In the following the abstract domains for the control flow analysis of Carmel are discussed in detail, including the techniques used for constructing the analysis domains as systematic abstractions of the corresponding concrete semantic domains.

**Remark 3.6 (Notation).** *Overlining is used as a notational convention to indicate abstract domains, e.g., $\overline{\mathsf{Ref}}$, often obtained by abstracting the corresponding concrete domain, e.g., the domain* $\mathsf{Ref}$. *In order for an analysis to be well-defined it is frequently necessary to require that certain abstract domains are equipped with additional structure, e.g., that domains are complete lattices. Here abstract domains which are also complete lattices are indicated as "hatted" domains, e.g., $\widehat{\mathsf{Stack}}$.*

For the control flow analysis numbers need not be tracked in much detail and thus numbers are simply represented as a single abstract value indicating "any number":

$$\overline{\mathsf{Num}} = \{\mathsf{INT}\}$$

In Section 4.1 a more precise abstract domain for numbers is introduced. That domain is then used to specify a *data flow analysis* for Carmel.

When designing a control flow analysis for an object-oriented language arguably one of the most important aspects to consider is how to track the flow of object references. For this reason the abstraction of object references should be considered and chosen carefully to ensure the usefulness of the abstraction. Due to the severe memory constraints that apply to Java Card applets, typical applets only create and instantiate new objects during the installation and initialisation phase. Furthermore, classes are only rarely instantiated more than once and thus it is often the case that only a single object of a given class exists during the lifetime of an applet, cf. [Che00, Mar00]. This leads naturally to the following abstract domain for object references:

$$\overline{\mathsf{ObjRef}} = \mathsf{Class} \uplus \{\mathtt{null}\}$$

Object references are simply abstracted into the class of the object that they reference. This is similar to the *class object graphs* defined in [VHU92]. An obvious improvement would be to use an abstraction similar to the *textual object graphs* also discussed in [VHU92]; essentially the textual object graphs abstract an object reference into its class and its creation point. However, as noted above, typical Java Card applets usually only instantiate a class once and in the case

where several objects of the same class are needed they are likely to be created at the same point in the program; thereby negating the advantage gained by using textual object graphs. For these reasons modelling object references as class object graphs is sufficient for the present analysis. In order to improve legibility abstract object references, $\sigma \in \mathsf{ObjRef}$, are written $(\mathsf{Ref}\ \sigma)$.

Array references are modelled, not unlike object references, simply by the type of elements an array contains:

$$\overline{\mathsf{ArrRef}} = \mathsf{ElemType}$$

Abstract array references are written $(\mathsf{Ref}\ (\texttt{array}\ t))$ rather than $t$ for readability. This particular choice of abstract domain for array references implies that all arrays containing elements of the same type are indistinguishable in the analysis. While this is sufficiently precise for the present work it would be straightforward to extend the above mentioned textual object graphs of [VHU92] to arrays and array references.

The abstract domain for references can now be defined:

$$\overline{\mathsf{Ref}} = \overline{\mathsf{ObjRef}} + \overline{\mathsf{ArrRef}}$$

Finally an abstract domain for return addresses, used for subroutines, is needed to complete the abstract domains for basic values. Because subroutines are local in a method return addresses can be modelled by program counters:

$$\overline{\mathsf{RetAddr}} = \mathbb{N}_0$$

An abstract return address, $pc \in \overline{\mathsf{RetAddr}}$, is written $(\mathsf{Adr}\ pc)$.

The above domains can now be combined to define the abstract domain for values:

$$\overline{\mathsf{Val}} = \overline{\mathsf{Num}} + \overline{\mathsf{Ref}} + \overline{\mathsf{RetAddr}}$$

For analysis purposes it is convenient to work with sets of such values and thus the following complete lattice is used as the basic domain of abstract values:

$$\widehat{\mathsf{Val}} = \mathcal{P}(\overline{\mathsf{Val}})$$

This domain admits the following ordering on abstract values using subset inclusion. Let $\hat{v}_1, \hat{v}_2 \in \widehat{\mathsf{Val}}$ and define:

$$\hat{v}_1 \sqsubseteq_{\widehat{\mathsf{Val}}} \hat{v}_2 \quad \text{iff} \quad \hat{v}_1 \subseteq \hat{v}_2$$

Proposition 3.9 below shows that this domain indeed is a complete lattice.

The state of an object in Carmel is simply the state of the objects instance fields; objects can therefore be modelled as maps from (instance) field to abstract values:

$$\widehat{\mathsf{Object}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$$

By extending the ordering on $\widehat{\mathsf{Val}}$, i.e., subset inclusion, in a point-wise manner an ordering on abstract objects is obtained. Let $\hat{o}_1, \hat{o}_2 \in \widehat{\mathsf{Object}}$ and define

$$\hat{o}_1 \sqsubseteq_{\widehat{\mathsf{Object}}} \hat{o}_2 \quad \text{iff} \quad \mathrm{dom}(\hat{o}_1) \subseteq \mathrm{dom}(\hat{o}_2) \ \wedge\ \forall f \in \mathrm{dom}(\hat{o}_1) : \hat{o}_1.f \subseteq \hat{o}_2.f$$

For the simple control flow analysis the length and structure of an array is not needed and is thus abstracted away:

$$\widehat{\mathsf{Array}} = \widehat{\mathsf{Val}}$$

The ordering on $\widehat{\mathsf{Val}}$ trivially induces the same ordering on abstract arrays.

The abstract domains defined thus far have all been targeted towards a *flow insensitive* analysis. This could easily be extended to the abstract domains for stacks and the local heap, i.e., modelling both simply as a set of abstract values. However, for several of the intended uses of the analysis more precision is required and in particular a locally, i.e., intra-procedurally, *flow sensitive* analysis is needed. This leads to an abstract representation of the local heap and the operand stack indexed over the addresses of the program. In other words, an abstract local heap and an abstract operand stack is associated to *every instruction* in the program. This is similar to the approach taken by Freund and Mitchell in [FM99, FM98]:

$$\widehat{\mathsf{LocHeap}} = \mathsf{Addr} \to \mathsf{Var} \to \widehat{\mathsf{Val}}$$

This domain admits an ordering: let $\hat{L}_1, \hat{L}_2 \in \widehat{\mathsf{LocHeap}}$ and define for all $(m_1, pc_1), (m_2, pc_2) \in \mathsf{Addr}$:

$$\begin{array}{l}\hat{L}_1 \sqsubseteq_{\widehat{\mathsf{LocHeap}}} \hat{L}_2 \quad \text{iff} \\ \quad \forall x \in \mathrm{dom}(\hat{L}_1(m_1, pc_1)) \colon \hat{L}_1(m_1, pc_1)(x) \subseteq \hat{L}_2(m_2, pc_2)(x)\end{array}$$

For succinctness and readability of the Flow Logic specification notational shorthands are introduced: $\hat{L}_1(m_1, pc_1) \sqsubseteq_Y \hat{L}_2(m_2, pc_2)$ with $Y \subseteq \mathsf{Var}$ defined as follows

$$\begin{array}{l}\hat{L}_1 \sqsubseteq_Y \hat{L}_2 \quad \text{iff} \\ \quad \forall x \in \mathrm{dom}(\hat{L}_1(m_1, pc_1)) \setminus Y \colon \hat{L}_1(m_1, pc_1)(x) \subseteq \hat{L}_2(m_2, pc_2)(x)\end{array}$$

and take $(A_0 :: \cdots :: A_n) \sqsubseteq \hat{L}(m, pc)[0..n]$ to stand for

$$\forall i \in \{0, \ldots, n\} \colon A_i \subseteq \hat{L}(m, pc)(i)$$

This is particularly useful when specifying the analysis for transferring actual parameters in method invocations.

Because the operand stack is used to transfer return values from method invocations an extended address domain is used, with a special program counter for every method explicitly representing the end of control flow for that method. It also facilitates the analysis of methods with more than one `return` instruction:

$$\overline{\mathsf{Addr}} = \mathsf{Addr} + (\mathsf{Method} \times \{\mathsf{END}\})$$

This corresponds to the usual requirement for control flow graphs to have special and explicit entry and exit nodes, cf. [NNH99, ASU85]. Since methods have a

unique and easily identifiable entry point, namely the instruction at program counter $pc = 0$, only a special end-point is needed.

The semantics of Carmel in principle allows stacks of infinite length to be constructed[1] and therefore the abstract representation must allow for this; thus a stack is abstractly modelled as a potentially infinite sequence of abstract values:

$$\widehat{\mathsf{Val}}^{\infty} = \widehat{\mathsf{Val}}^{\omega} \cup \widehat{\mathsf{Val}}^{*}$$

where $\widehat{\mathsf{Val}}^{\omega}$ is the set of all infinite sequences of abstract values and $\widehat{\mathsf{Val}}^{*}$ is the set of all finite sequences defined by

$$\widehat{\mathsf{Val}}^{*} = \bigcup_{i \geq 0}^{\infty} \widehat{\mathsf{Val}}^{i}$$

with $\widehat{\mathsf{Val}}^{0} = \emptyset$, $\widehat{\mathsf{Val}}^{1} = \widehat{\mathsf{Val}}$, and $\widehat{\mathsf{Val}}^{i+1} = \widehat{\mathsf{Val}} \times \widehat{\mathsf{Val}}^{i}$. Elements of $\widehat{\mathsf{Val}}^{\infty}$ are written as $S_1 :: \cdots :: S_n$ for finite sequences, $S_1 :: S_2 :: \cdots$ for infinite sequences and $\epsilon$ for the empty sequence. The length of such elements is defined as follows

$$|S| = \begin{cases} 0 & \text{if } S = \epsilon \\ n & \text{if } S = (S_1 :: \cdots :: S_n) \in \widehat{\mathsf{Val}}^{*} \\ \infty & \text{if } S \in \widehat{\mathsf{Val}}^{\omega} \end{cases}$$

Thus an ordering can be imposed on $\widehat{\mathsf{Val}}^{\infty}$. Writing $S|_i$ for the $i$th element of $S \in \widehat{\mathsf{Val}}^{\infty}$ with $|S| \geq i$ the following ordering relation is defined

$$S \sqsubseteq_{\widehat{\mathsf{Val}}^{\infty}} T \quad \text{iff} \quad |S| \leq |T| \ \wedge \ \forall i : 1 \leq i \leq |S| : S|_i \subseteq T|_i$$

While the domain $\widehat{\mathsf{Val}}^{\infty}$ is certainly sufficient for modelling stacks that arise during the evaluation of a Carmel program, it is rather inconvenient for the intended applications and the developments in subsequent chapters where infinite stacks are not possible. Therefore a very similar but slightly simpler domain is used, with essentially the same ordering as for $\widehat{\mathsf{Val}}^{\infty}$ and using a top-element to represent infinite stacks: $(\widehat{\mathsf{Val}}^{*})^{\top}$. Rather than introduce this in an ad-hoc manner a *Galois connection*, a concept well-known from *abstract interpretation*, can be used to systematically relate the above two domains. Formally a Galois connections is defined as follows:

**Definition 3.7 (Galois connection).** *For the partially ordered sets $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$, the pair $(\alpha, \gamma)$ defines a* Galois connection*, written $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$, if and only if*

1. *$\alpha : L \to M$ and $\gamma : M \to L$ are monotone functions*
2. *$id_L \sqsubseteq_L \gamma \circ \alpha$*
3. *$\alpha \circ \gamma \sqsubseteq_M id_M$*

[1] The bytecode verifier rejects all programs that may give rise to infinite stacks.

*where $id_L$ and $id_M$ are the identity functions on $L$ and $M$ respectively.*

As mentioned, Galois connections play an important role in the field of abstract interpretation, cf. [CC77, NNH99], where they are used to systematically construct simpler and more abstract domains from complex and concrete domains. In abstract interpretation Galois connections are (usually) used to relate the concrete domains to their abstract counterparts and thereby prove the correctness of the abstraction and thus the correctness of the analysis. Here Galois connections are used to validate the use of a less obvious but more practical abstract domain instead of the more obvious (but less practical) abstract domain $\widehat{\mathsf{Val}}^\infty$. A more systematic use of Galois connections in the Flow Logic framework is certainly feasible but the usefulness of such an approach for the present work is less evident, cf. [NHN03, HJNN99].

For the present purpose an *abstraction function* is defined on $\widehat{\mathsf{Val}}^\infty$ that formalises the above discussion by mapping infinite stacks to the top element of $(\widehat{\mathsf{Val}}^*)^\top$ and acts as the identity map on finite stacks:

$$\alpha_{\mathsf{Stack}}(S) = \begin{cases} S & \text{if } S \in \widehat{\mathsf{Val}}^* \\ \top_{\mathsf{Stack}} & \text{if } S \in \widehat{\mathsf{Val}}^\omega \end{cases}$$

this leads to the following *conretisation function*:

$$\gamma_{\mathsf{Stack}}(S) = \begin{cases} S & \text{if } S \neq \top_{\mathsf{Stack}} \\ \widehat{\mathsf{Val}} :: \cdots & \text{if } S = \top_{\mathsf{Stack}} \end{cases}$$

where $\widehat{\mathsf{Val}} :: \cdots$ is an infinite sequence with each element identical to the entire set of $\widehat{\mathsf{Val}}$.

The above functions can now be used to state and prove the formal relationship between the two domains for operand stacks:

**Proposition 3.8.** *The pair $(\alpha_{\mathsf{Stack}}, \gamma_{\mathsf{Stack}})$ defines a Galois connection:*

$$\widehat{\mathsf{Val}}^\infty \underset{\alpha_{\mathsf{Stack}}}{\overset{\gamma_{\mathsf{Stack}}}{\leftrightarrows}} (\widehat{\mathsf{Val}}^*)^\top$$

*Proof.* It is trivial to show that $\alpha_{\mathsf{Stack}}$ and $\gamma_{\mathsf{Stack}}$ are monotone functions. In order to prove that $\gamma_{\mathsf{Stack}} \circ \alpha_{\mathsf{Stack}} \sqsupseteq id$ the proof is split into two cases. First let $S \in \widehat{\mathsf{Val}}^*$ then by definition

$$\begin{array}{rcl} \gamma_{\mathsf{Stack}}(\alpha_{\mathsf{Stack}}(S)) & = & \gamma_{\mathsf{Stack}}(S) \\ & = & S \\ & \sqsupseteq & S \end{array}$$

Now assume that $S \in \widehat{\mathsf{Val}}^\omega$ it then follows that

$$\begin{array}{rcl} \gamma_{\mathsf{Stack}}(\alpha_{\mathsf{Stack}}(S)) & = & \gamma_{\mathsf{Stack}}(\top_{\mathsf{Stack}}) \\ & = & \widehat{\mathsf{Val}} :: \cdots \\ & \sqsupseteq & S \end{array}$$

and thus $\gamma_{\mathsf{Stack}} \circ \alpha_{\mathsf{Stack}} \sqsupseteq id$. Similarly it can be shown that $\alpha_{\mathsf{Stack}} \circ \gamma_{\mathsf{Stack}} \sqsubseteq id$. It follows that $(\alpha_{\mathsf{Stack}}, \gamma_{\mathsf{Stack}})$ is a Galois connection. ∎

Finally the abstract domain for stacks can be defined in the following way:

$$\widehat{\mathsf{Stack}} = \overline{\mathsf{Addr}} \to (\widehat{\mathsf{Val}}^*)^\top$$

In accordance with the concrete domain for heaps being split into a static and a dynamic component, two separate abstract domains are defined to model the static and dynamic heap respectively. The static component holds the values of all static fields and is modelled in a straightforward manner as a map from (static) fields to a set of abstract values:

$$\widehat{\mathsf{StaHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$$

The point-wise extension of the ordering on $\widehat{\mathsf{Val}}$ yields an ordering on $\widehat{\mathsf{StaHeap}}$:

$$\hat{K}_1 \sqsubseteq_{\widehat{\mathsf{StaHeap}}} \hat{K}_2 \quad \text{iff} \quad \forall f \in \mathrm{dom}(\hat{K}_1) : \hat{K}_1.f \subseteq \hat{K}_2.f$$

The dynamic component of the heap contains the objects and arrays that are created during program execution, both of which must be accommodated in the abstract representation:

$$\widehat{\mathsf{Heap}} = (\overline{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}) \times (\overline{\mathsf{ArrRef}} \to \widehat{\mathsf{Array}})$$

For an abstract heap $\hat{H}$ the notation $\hat{H}(\mathsf{Ref}\ \sigma)$ is used for $\pi_1(\hat{H})(\mathsf{Ref}\ \sigma)$ and similarly $\pi_2(\hat{H})(\mathsf{Ref}\ (\mathtt{array}\ t))$ is written $\hat{H}(\mathsf{Ref}\ (\mathtt{array}\ t))$ where $\pi_1$ and $\pi_2$ are the projections of the first and second component respectively. The notaion is well-defined because $\overline{\mathsf{ObjRef}} \cap \overline{\mathsf{ArrRef}} = \emptyset$.

This leads to the following ordering induced from the ordering on abstract objects and arrays respectively. For $\hat{H}_1, \hat{H}_2 \in \widehat{\mathsf{Heap}}$:

$$\begin{aligned}\hat{H}_1 \sqsubseteq_{\widehat{\mathsf{Heap}}} \hat{H}_2 \quad \text{iff} \quad & \forall(\mathsf{Ref}\ \sigma) \in \mathrm{dom}(\hat{H}_1) : \hat{H}_1(\mathsf{Ref}\ \sigma) \sqsubseteq_{\widehat{\mathsf{Object}}} \hat{H}_2(\mathsf{Ref}\ \sigma) \wedge \\ & \forall(\mathsf{Ref}\ (\mathtt{array}\ t)) \in \mathrm{dom}(\hat{H}_1) : \\ & \quad \hat{H}_1(\mathsf{Ref}\ (\mathtt{array}\ t)) \subseteq \hat{H}_2(\mathsf{Ref}\ (\mathtt{array}\ t))\end{aligned}$$

Note that the notational conventions imply that $\forall(\mathsf{Ref}\ \sigma) \in \mathrm{dom}(\hat{H}_1)$ is equivalent to $\forall\sigma \in \mathrm{dom}(\pi_1(\hat{H}_1))$ and similarly $(\mathsf{Ref}\ (\mathtt{array}\ t)) \in \mathrm{dom}(\hat{H}_1)$ is equivalent to $\forall t \in \mathrm{dom}(\pi_2(\hat{H}_1))$.

The domain for the control flow analysis can now be defined by combining the abstract domains for the global heap (both dynamic and static), the local heap, and an operand stack:

$$\widehat{\mathsf{Analysis}}_{\mathsf{CFA}} = \widehat{\mathsf{StaHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocHeap}} \times \widehat{\mathsf{Stack}}$$

It should be evident that elements of the above domain contain the abstract information equivalent to a semantic configuration, which is indeed what must be proved formally in order to establish the semantic correctness of the analysis.

A point-wise ordering, $\sqsubseteq_{\mathsf{CFA}}$, can trivially be defined on the analysis domain:

$$\begin{aligned}&(\hat{K}_1, \hat{H}_1, \hat{L}_1, \hat{S}_1) \sqsubseteq_{\mathsf{CFA}} (\hat{K}_2, \hat{H}_2, \hat{L}_2, \hat{S}_2) \quad \text{iff} \\ &\quad \hat{K}_1 \sqsubseteq_{\widehat{\mathsf{StaHeap}}} \hat{K}_2 \ \wedge\ \hat{H}_1 \sqsubseteq_{\widehat{\mathsf{Heap}}} \hat{H}_2 \ \wedge\ \hat{L}_1 \sqsubseteq_{\widehat{\mathsf{LocHeap}}} \hat{L}_2 \ \wedge\ \hat{S}_1 \sqsubseteq_{\widehat{\mathsf{Stack}}} \hat{S}_2\end{aligned}$$

This ordering makes it possible to compare two analysis result for a given program and indeed to determine which of the two is the smallest solution with respect to the ordering.

The following proposition formally states and proves that a number of the above mentioned domains are indeed complete lattices. Note that the same name is used for both the lattice and its carrier set, e.g., $\widehat{\mathsf{Val}} = (\widehat{\mathsf{Val}}, \subseteq)$.

**Proposition 3.9.** *The following domains define complete lattices:*

1. $\widehat{\mathsf{Val}} = (\widehat{\mathsf{Val}}, \subseteq)$
2. $\widehat{\mathsf{Array}} = (\widehat{\mathsf{Array}}, \subseteq)$
3. $\widehat{\mathsf{Object}} = (\widehat{\mathsf{Object}}, \sqsubseteq_{\widehat{\mathsf{Object}}})$
4. $\widehat{\mathsf{LocHeap}} = (\widehat{\mathsf{LocHeap}}, \sqsubseteq_{\widehat{\mathsf{LocHeap}}})$
5. $\widehat{\mathsf{Stack}} = (\widehat{\mathsf{Stack}}, \sqsubseteq_{\widehat{\mathsf{Stack}}})$
6. $\widehat{\mathsf{StaHeap}} = (\widehat{\mathsf{StaHeap}}, \sqsubseteq_{\widehat{\mathsf{StaHeap}}})$
7. $\widehat{\mathsf{Heap}} = (\widehat{\mathsf{Heap}}, \sqsubseteq_{\widehat{\mathsf{Heap}}})$
8. $\widehat{\mathsf{Analysis}}_{\mathsf{CFA}} = (\widehat{\mathsf{Analysis}}_{\mathsf{CFA}}, \sqsubseteq_{\mathsf{CFA}})$

*Proof.* Cases 1 and 2 follow trivially from [DP90, Example 2.7(2)]. The remaining cases follow from [DP90, Paragraph 2.14] since the orderings introduced are simply the pointwise extension of the ordering on $\widehat{\mathsf{Val}}$, namely $\subseteq$. ∎

In the sequel the subscript of the ordering relation is omitted when is it clear from the context.

### 3.2.3 Flow Logic Specification

Having defined the universe of discourse, i.e., the abstract domains, for the control flow analysis, the analysis itself can now be specified. As discussed in Section 3.1 this is done by specifying judgements that define when a proposed analysis result is acceptable with respect to a given program. For the control flow analysis of Carmel the judgements are on the form:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\mathrm{CFA}} (m, pc) : \texttt{instr}$$

where $(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \in \widehat{\mathsf{Analysis}}_{\mathsf{CFA}}$, and `instr` is the instruction at program counter $pc$ in method $m$. Intuitively the judgement states that $(\hat{K}, \hat{H}, \hat{L}, \hat{S})$ is an *acceptable analysis* of the instruction `instr` at address $(m, pc)$.

In the following the judgements for a few specific instructions are discussed in depth to introduce both the notation and conventions used, and also to give

some insight into how the analysis and methodology works. To facilitate the Flow Logic specification some special notation for variable binding is introduced:

$$A \triangleleft B :$$

meaning that the value of $B$ is bound to the variable $A$ that may then be referenced later. By extending the notation slightly with a simple form of pattern matching it becomes particularly useful for manipulating abstract stacks. For example

$$A_1 :: \cdots :: A_n :: X \triangleleft \hat{S}(m, pc) :$$

means that the abstract stack at $\hat{S}(m, pc)$ must contain at least $n$ elements, bound to variables $A_1$ through $A_n$ respectively. The rest of the stack (if any) is then bound to $X$. The elements of the stack can then be conveniently referenced using the variables making the specification easer to read (and write).

### The `push` Instruction

The Flow Logic judgement for the `push` instruction is of the following form:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{push}\ \ t\ \ v$$

From the semantics it is known that `push` simply pushes its argument $v$ on top of the current stack. In the analysis the stack at address $(m, pc)$ is contained in $\hat{S}(m, pc)$ and thus the effect of push is to create a new stack with the abstract representation of $v$, denoted $\beta_{\mathsf{Const}}(v)$ here, on top of the current stack: $\beta_{\mathsf{Const}}(v) :: \hat{S}(m, pc)$; this is then available at the next instruction which is located at $(m, pc + 1)$. This is formulated as follows

$$\beta_{\mathsf{Const}}(v) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$$

where

$$\beta_{\mathsf{Const}}(c) = \begin{cases} \{\mathsf{INT}\} & \text{if } c \in \mathsf{Num} \\ \{\texttt{null}\} & \text{if } c = \texttt{null} \\ \{(\mathsf{Adr}\ c)\} & \text{if } c \in \mathsf{RetAdr} \end{cases}$$

In Section 3.3 it will become evident that the $\beta_{\mathsf{Const}}$ function is a special case of the *representation function* for values.

No local variables are modified by executing a `push` instruction and thus the current local heap is just copied forward to the next instruction without change:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$

Combining the above formulae results in the following specification for the `push` instruction:

$$\begin{array}{l} (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{push}\ \ t\ \ c \\ \quad \text{iff} \quad \beta_{\mathsf{Const}}(v) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1) \\ \qquad\quad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1) \end{array}$$

Note that since both the static heap and the dynamic heap are defined as global components, contrary to the local heap and the operand stack, it is not necessary to explicitly copy them forward when they are not modified.

### The `store` Instruction

The next instruction of interest is the `store` instruction:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{store}\ t\ x$$

This instruction saves the top element of the stack in the variable given as argument, $x$, to the instruction. Clearly, in order for this to work properly there must be at least one element on the stack which is checked by the bytecode verification. The situation is similar in the analysis and special notation is introduced to aid the specification:

$$A :: X \triangleleft \hat{S}(m, pc) :$$

This notation provides a succinct way of expressing that the abstract stack should be of the form $A :: X$, i.e., it should have at least one element. Furthermore, it acts as a binder for the variables $A$ and $X$ so that they may subsequently be referred to in a convenient manner.

Storing the top element of the stack, denoted $A$, in the local heap, available for the next instruction, is modelled as:

$$A \sqsubseteq \hat{L}(m, pc+1)(x)$$

Similarly having stored and thus popped the top of the stack, the bottom of the stack is then copied forward to the next instruction:

$$X \sqsubseteq \hat{S}(m, pc+1)$$

Finally, the local variables that were not modified by the instruction, i.e., all but $x$, must be transferred to the next instruction:

$$\hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1)$$

Putting the above together results in the following clause for `store` instructions:

$$\begin{array}{rl} (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{store}\ t\ x & \\ \quad \text{iff} & A :: X \triangleleft \hat{S}(m, pc) : \\ & A \sqsubseteq \hat{L}(m, pc+1)(x) \\ & X \sqsubseteq \hat{S}(m, pc+1) \\ & \hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1) \end{array}$$

### The `new` Instruction

The judgement for the `new`-instruction follows the same pattern:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{new}\ \sigma$$

The `new`-instruction allocates room on the heap for a new instance of the class given as argument to the instruction and returns a reference to the object on top of the stack:

$$\{(\mathsf{Ref}\ \sigma)\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$$

The fields of the new object are all initialised with their default value: 0 for numeric types and `null` for reference types:

$$\mathit{default}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)$$

where *default* is defined as follows:

$$\forall f \in \mathit{fields}(\sigma):\ \ \mathit{default}(\sigma)(f) = \beta_{\mathsf{Const}}(\mathit{def}(f.\mathit{type}))$$

Note that *def* is the function, also used in the semantics, that maps types to their default values, i.e., numeric types to 0 and reference types to `null`. The representation function, $\beta_{\mathsf{Const}}$, also used for the `push`-instruction, is used here to map the default values to their abstract representations.

No local variables were modified and therefore the local heap is simply copied forward:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$

Combining the above gives the following clause for `new`-instructions:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{new}\ \ \sigma \\ \quad \text{iff} \quad \{(\mathsf{Ref}\ \sigma)\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1) \\ \qquad\ \ \mathit{default}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma) \\ \qquad\ \ \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)\end{array}$$

### The `putfield` Instruction

Next is the specification for `putfield` instructions:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putfield}\ \ f$$

The `putfield` instruction stores the first element of the stack in the field given as argument to the instruction. The second element of the stack must be a reference to the specific object in whose field the value should be stored. Thus the stack must contain at least two elements:

$$A :: B :: X \triangleleft \hat{S}(m, pc) :$$

To model the concrete behaviour in the analysis all the elements on top of the stack are copied to *all* the objects referenced in the second element of the stack:

$$\forall(\mathsf{Ref}\ \sigma) \in B:\ A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f)$$

As already noted objects are abstracted into their class and thus the contents of the fields of objects of the same class is also merged and stored in the abstract heap.

The bottom of the stack is then copied forward to the next instruction:

$$X \sqsubseteq \hat{S}(m, pc + 1)$$

and since no local variables were modified, the abstract local heap is transferred unchanged to the next instruction:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$

Thus the combined clause for `putfield`-instructions:

$$\begin{array}{rl} (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putfield } f & \\ \quad \text{iff} & A :: B :: X \triangleleft \hat{S} : \\ & \forall(\mathsf{Ref}\ \sigma) \in B\colon\ A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f) \\ & X \sqsubseteq \hat{S}(m, pc + 1) \\ & \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1) \end{array}$$

**The `invokevirtual` Instruction**

The analysis of `invokevirtual` is discussed:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual } m_0$$

The arguments for the method must be found on top of the stack and, as for the `putfield` instruction, a reference to the target object must be found on the stack immediately following the arguments. Using $|m_0|$ to denote the arity of method $m_0$ (note that $|m_0|$ is statically determined by the type of $m_0$) the required stack layout can be specified:

$$A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) :$$

Following the semantics, the *methodLookup* function is used directly in the analysis of the dynamic dispatch of virtual methods. This is possible because the *methodLookup* function is really a representation of the class-hierarchy in a program and the class-hierarchy does not change dynamically during run-time and can therefore be computed at compile-time.

Because of the approximative nature the analysis there may be more than one object reference to the target object(s). Therefore it may be necessary to do method lookups for several references:

$$\begin{array}{l} \forall(\mathsf{Ref}\ \sigma) \in B\colon \\ \quad m_v \triangleleft \mathit{methodLookup}(m_0, \sigma) : \ldots \end{array}$$

Again the '$\ldots\triangleleft\ldots$' notation is used to bind the value of the right-hand side (here the looked up method) to the variable on the left-hand side for later reference.

For each of the looked up methods, $m_v$, the arguments must now be transferred as *local variables* in the invoked method, available in the very first instruction, i.e., with $pc = 0$. Furthermore a reference to the object containing the invoked method, the `self` reference, must be passed to the invoked method in local variable number 0:

$$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$$

Note that the semantics of inheritance and virtual methods in Carmel (and JCVML) requires that $m_0$ and $m_v$ are of the same type, i.e., accept the same arguments and return a value of the same type. In particular this entails that $|m_v| = |m_0|$.

When an invoked method returns it may or may not return a value. In the latter case the invoked method has return type `void`. In this case the bottom of the stack should simply be copied forward to the next instruction:

$$\begin{array}{l} m_0.returnType = \texttt{void} \Rightarrow \\ \quad X \sqsubseteq \hat{S}(m, pc+1) \end{array}$$

In the former case the invoked method has a return type different from `void` and the return value can then be found at the top of the stack of the *invoked* method, $m_v$, at the special address: $(m_v, \mathsf{END})$ used to indicate the logical end of a method. The value found there must then be copied back to the top of the stack of the *invoking* method (less the arguments and the object reference that are popped off the stack) and passed forward to the next instruction:

$$\begin{array}{l} m_0.returnType \neq \texttt{void} \Rightarrow \\ \quad A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : \ A :: X \sqsubseteq \hat{S}(m, pc+1) \end{array}$$

Finally, since none of the local variable of the invoking method have been changed they are copied forward unchanged:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$$

The resulting combined clause for `invokevirtual`:

$$\begin{array}{l} (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual}\ m_0 \\ \quad \text{iff} \quad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) : \\ \qquad \forall(\mathsf{Ref}\ \sigma) \in B : \\ \qquad\quad m_v \triangleleft methodLookup(m_0, \sigma) : \\ \qquad\quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|] \\ \qquad\quad m_0.returnType = \texttt{void} \Rightarrow \\ \qquad\qquad X \sqsubseteq \hat{S}(m, pc+1) \\ \qquad\quad m_0.returnType \neq \texttt{void} \Rightarrow \\ \qquad\qquad A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : \ A :: X \sqsubseteq \hat{S}(m, pc+1) \\ \qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \end{array}$$

### The `arrayload` Instruction

The instructions for manipulating arrays are very straightforward. Here the specification of the `arrayload`-instruction is discussed:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{arrayload}\ t$$

The `arrayload` instruction takes two arguments: a reference to the array and an index into the array; it then returns the value of the referenced array at

the given index on top of the stack for the next instruction. Since arrays are modelled as the set of (abstract) values the array can possibly contain the entire (abstract) array is simply copied forward to the next instruction:

$$\begin{array}{l}\forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B: \\ \quad \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)) :: X \sqsubseteq \hat{S}(m, pc+1)\end{array}$$

This instruction does not modify local variables and thus the local heap is just carried forward:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$$

The analysis of the `arrayload`-instruction can thus be specified as follows:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{arrayload}\ t \\ \quad \text{iff} \quad A :: B :: X \triangleleft \hat{S}(m, pc) : \\ \qquad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B: \\ \qquad\quad \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)) :: X \sqsubseteq \hat{S}(m, pc+1) \\ \qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)\end{array}$$

### The `jsr` and `ret` Instructions

Subroutines are mostly used to implement the `try/finally` functionality of Java Card that is particularly useful for executing "clean-up" code in the event of exceptional control flow.

The `jsr`-instruction initiates a subroutine and works almost like a `goto` except that the address of the instruction following the `jsr`-instruction is put on top of the stack. The intention is then that the called subroutine should save this address in a local variable and then use it later for jumping back to where the subroutine was called from. Thus subroutines can be seen as an advanced form of the `goto`-instruction. This is modelled straightforwardly in the analysis:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{jsr}\ \ pc_0 \\ \quad \text{iff} \quad \{(\mathsf{Adr}\ pc+1)\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc_0) \\ \qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)\end{array}$$

To return from a subroutine the `ret` instruction is used: it jumps (back) to the program counter stored in the local variable given as an argument to the instruction. In the analysis care is taken to copy the current stack and local heap to all the possible return addresses:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{ret}\ \ i \\ \quad \text{iff} \quad \forall(\mathsf{Adr}\ pc_0) \in \hat{L}(m, pc)(i): \\ \qquad\quad \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc_0) \\ \qquad\quad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)\end{array}$$

The conceptual simplicity of subroutines hides a subtle issue with type checking of subroutines: the local variables not used by a given subroutine may contain

values of different, conflicting, types depending on where it was called from. In order to deal with this the type system has to incorporate a limited form of polymorphism, cf. [FM99, Fre98]. Since a control flow analysis is only concerned with tracking (or approximating) the actual control flow of a program and not with prohibiting "illegal" or unwanted control flow, the analysis of subroutines is straightforward and does not require any special considerations. Thereby the program analysis approach sidesteps some of the modelling problems brought about by subroutine polymorphism, cf. [Fre98].

### 3.2.4 Control Flow Analysis: Full Specification

The Flow Logic specification for the control flow analysis of the full Carmel language is divided into six fragments like the semantics: imperative core (Figures 3.1, 3.2, and 3.3), objects (Figure 3.4), methods (Figure 3.5), arrays (Figure 3.6), and subroutines (Figure 3.7). Exceptions are not covered by the basic control flow analysis; however, an exception analysis is developed in Section 3.5.

### 3.2.5 Analysing Programs

Taking initial configurations into account, cf. Definition 2.1, the Flow Logic specification is extended to cover the analysis of entire programs as follows:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} P \quad \text{iff} \\
\quad \forall (m, pc) \in P.\mathit{addresses}: \\
\qquad m.\mathit{instructionAt}(pc) = \texttt{instr} \Rightarrow (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{instr} \\
\quad \forall \sigma \in P.\mathit{main}: \\
\qquad \mathit{default}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma) \\
\qquad m_\sigma = \sigma.\mathit{entry} \Rightarrow (\mathsf{Ref}\ \sigma) \in \hat{L}_0(m_\sigma, 0)
\end{array}
$$

where $P.\mathit{addresses}$ is the set of addresses occurring in the program $P$ defined as follows:

$$
\begin{array}{r}
P.\mathit{addresses} = \{(m, pc) \mid pc \in \mathrm{dom}(m.\mathit{instructionAt}), m \in \sigma.\mathit{methods}, \\
\sigma \in p.\mathit{classes}, p \in P.\mathit{packages}\}
\end{array}
$$

## 3.3 Theoretical Properties

In this section two important theoretical properties are formally stated and proved for the control flow analysis namely: *semantic soundness* and a *Moore family* property for analysis results.

Semantic soundness proves that the analysis results can be trusted to correctly reflect the actual run-time behaviour of an analysed program and that the specified analysis does indeed analyse the intended property, in this case control flow. A formally stated and proved semantic soundness result is crucial when using the analysis for high-assurance purposes such as certifying that a program

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{nop}$
iff *true*

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{push}\ t\ c$
iff $\beta_{\mathsf{Const}}(v) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{pop}\ n$
iff $A_1 :: \cdots :: A_n :: X \triangleleft \hat{S}(m, pc) :$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{dup}\ m\ n$
iff $S_1 :: S_2 :: X \triangleleft \hat{S}(m, pc) :$
$A_1 :: \cdots :: A_m \triangleleft S_1 :$
$A_{m+1} :: \cdots :: A_n \triangleleft S_2 :$
$S_1 :: S_2 :: S_1 :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{swap}\ m\ n$
iff $S_1 :: S_2 :: X \triangleleft \hat{S}(m, pc) :$
$A_1 :: \cdots :: A_m \triangleleft S_1 :$
$A_{m+1} :: \cdots :: A_n \triangleleft S_2 :$
$S_2 :: S_1 :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{numop}\ t\ \mathit{unop}\ [t']$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{numop}\ t\ \mathit{binop}\ [t']$
iff $A_1 :: A_2 :: X \triangleleft \hat{S}(m, pc) :$
$\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

Figure 3.1: Imperative Core (1)

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{goto}\ pc_0$
iff $\hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc_0)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{if}\ t\ cmp\ \texttt{goto}\ pc_0$
iff $A_1 :: A_2 :: X \triangleleft \hat{S}(m, pc) :$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$X \sqsubseteq \hat{S}(m, pc_0)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{if}\ t\ cmp\ nul\ \texttt{goto}\ pc_0$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$X \sqsubseteq \hat{S}(m, pc_0)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{lookupswitch}\ t\ (k_i\texttt{=>}pc_i)_1^n\ \texttt{default=>}pc_{n+1}$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\forall i \in \{1, \ldots, n, n + 1\} :$
$X \sqsubseteq \hat{S}(m, pc_i)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_i)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{tableswitch}\ t\ l\texttt{=>}(pc_i)_0^n\ \texttt{default=>}pc_{n+1}$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\forall i \in \{0, \ldots, n, n + 1\} :$
$X \sqsubseteq \hat{S}(m, pc_i)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_i)$

Figure 3.2: Imperative Core (2)

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{load}\ t\ x$$
$$\text{iff} \quad \hat{L}(m, pc)(x) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$$
$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{store}\ t\ x$$
$$\text{iff} \quad A :: X \triangleleft \hat{S}(m, pc) :$$
$$A \sqsubseteq \hat{L}(m, pc+1)(x)$$
$$X \sqsubseteq \hat{S}(m, pc+1)$$
$$\hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1)$$

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{inc}\ t\ x\ n$$
$$\text{iff} \quad \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$$
$$\{\mathsf{INT}\} \sqsubseteq \hat{L}(m, pc+1)(x)$$
$$\hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1)$$

Figure 3.3: Imperative Core (3)

does not exhibit certain unwanted behaviours as for example is required by the higher assurance levels (EAL5 and above) of the Common Criteria [CC99].

The Moore family property, also proved in the following, serves two purposes: first and foremost for showing that every program can be analysed, and second, that there exists a *smallest* or *best* analysis for every program.

### 3.3.1 Semantic Soundness

In this section the semantic soundness of the control flow analysis is proved. This is done, as is usual in the Flow Logic framework when working with a small-step semantics, by establishing a *subject reduction* property for the analysis. The technique is well-known and often used in conjunction with proving the correctness of type systems, cf. [Pie02].

As a prerequisite for proving the subject reduction property the formal relationship between the concrete semantic domains and their corresponding abstract counterparts must established. Following [NNH99] this is done using *representation functions* and *correctness relations* that provide a very systematic and structured approach to formalising, stating, and proving the semantic soundness of analyses.

#### Representation Functions and Correctness Relations

Intuitively a representation function maps a concrete value, e.g., a number, to the *best possible* abstract representation in the corresponding abstract domain. For numbers the abstract domain is a singleton set and the representation function for numbers is therefore the constant function that maps every (concrete)

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{new}\ \sigma$
iff $\{(\mathsf{Ref}\ \sigma)\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\mathit{default}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{checkcast}\ t$
iff $\hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{instanceof}\ \sigma$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{getstatic}\ f$
iff $\hat{K}(f) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putstatic}\ f$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$A \sqsubseteq \hat{K}(f)$
$X \sqsubseteq \hat{S}(m, pc)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putfield}\ f$
iff $A :: B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B\colon A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f)$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putfield this}\ f$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in \hat{L}(m, pc)(0)\colon A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f)$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

Figure 3.4: Object Fragment

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokestatic}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: X \triangleleft \hat{S}(m, pc) :$
$A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_0, 0)[0..|m| - 1]$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_0, \mathsf{END}) :\ A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B :$
$m_v \triangleleft methodLookup(m_0, \sigma) :$
$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) :\ A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokeinterface}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B :$
$m_v \triangleleft methodLookup(m_0, \sigma) :$
$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) :\ A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{return}$
iff *true*

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{return}\ t$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$A :: \epsilon \sqsubseteq \hat{S}(m_0, \mathsf{END})$

Figure 3.5: Method Fragment

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{new (array } t\texttt{)} \\
\quad \text{iff} \quad A :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \{(\mathsf{Ref}\ (\texttt{array}\ t))\} :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{arraylength} \\
\quad \text{iff} \quad B :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \{\mathsf{INT}\} :: \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{arrayload } t \\
\quad \text{iff} \quad A :: B :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B\text{:} \\
\qquad\quad \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)) :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{arraystore } t \\
\quad \text{iff} \quad A_1 :: A_2 :: B :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B\text{:} \\
\qquad\quad A_1 \sqsubseteq \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)) \\
\qquad\quad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)
\end{array}
$$

Figure 3.6: Arrays

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{jsr } pc_0 \\
\quad \text{iff} \quad \{(\mathsf{Adr}\ pc+1)\} :: \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc_0) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc_0)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{ret } i \\
\quad \text{iff} \quad \forall(\mathsf{Adr}\ pc_0) \in \hat{L}(m,pc)(i)\text{:} \\
\qquad\quad \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc_0) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc_0)
\end{array}
$$

Figure 3.7: Subroutines

number to the abstract token INT:

$$\beta_{\mathsf{Num}}(n) = \{\mathsf{INT}\} \qquad \text{for } n \in \mathsf{Num}$$

The abstract representation of locations (references) depends on the class or type of object or array respectively that the location points to and thus the meaning of a location is always relative to a given heap. Therefore the representation function for locations is parameterised on a heap: $H \in \mathsf{Heap}$. Let $loc \in \mathrm{dom}(H)$ and define the representation function for locations as follows:

$$\beta^H_{\mathsf{Ref}}(loc) = \begin{cases} \{\mathtt{null}\} & \text{if } loc = \mathtt{null} \\ \{(\mathsf{Ref}\ \sigma)\} & \text{if } H(loc) \in \mathsf{Object} \text{ and } H(loc).class = \sigma \\ \{(\mathsf{Ref}\ (\mathtt{array}\ t))\} & \text{if } H(loc) \in \mathsf{Array} \text{ and } H(loc).type = t \end{cases}$$

To represent return addresses they are simply inject into the corresponding abstract domain:

$$\beta_{\mathsf{RetAdr}}(pc) = \{(\mathsf{Adr}\ pc)\} \qquad \text{for } pc \in \mathsf{RetAdr}$$

Using the above, a representation function for all basic values in the semantics, $v \in \mathsf{Val}$, is defined:

$$\beta^H_{\mathsf{Val}}(v) = \begin{cases} \beta_{\mathsf{Num}}(v) & \text{if } v \in \mathsf{Num} \\ \beta^H_{\mathsf{Ref}}(v) & \text{if } v \in \mathsf{Ref} \\ \beta_{\mathsf{RetAdr}}(v) & \text{if } v \in \mathsf{RetAdr} \end{cases}$$

Now the $\beta_{\mathsf{Const}}$ function defined in Section 3.2.3 can be seen as a special case of $\beta_{\mathsf{Val}}$ for values that does not depend on the heap:

$$\beta_{\mathsf{Const}}(c) = \begin{cases} \beta_{\mathsf{Num}}(c) & \text{if } c \in \mathsf{Num} \\ \{\mathtt{null}\} & \text{if } c = \mathtt{null} \\ \beta_{\mathsf{RetAdr}}(c) & \text{if } c \in \mathsf{RetAdr} \end{cases}$$

This representation function was used in the Flow Logic specification for the `push` instruction.

Stacks are finite sequences of values and are represented as sequences of corresponding abstract values obtained by using the representation function for basic values. For $S = v_1 :: \cdots :: v_n \in \mathsf{Stack}$ define:

$$\beta^H_{\mathsf{Stack}}(S) = \beta^H_{\mathsf{Val}}(v_1) :: \cdots :: \beta^H_{\mathsf{Val}}(v_n)$$

Since the local heap is implemented as a map from local variables to values, a representation function for local heaps can be obtained by simply composing the representation function for values with the local heap. For $L \in \mathsf{LocHeap}$ define

$$\beta^H_{\mathsf{LocHeap}}(L) = \beta^H_{\mathsf{Val}} \circ L$$

Abstract objects are represented as maps from fields to abstract values. Thus to map a concrete object into an abstract object it is sufficient to map the

contents of the concrete objects fields into the corresponding abstract value(s). For $o \in \mathsf{Object}$ define:

$$\beta^H_{\mathsf{Object}}(o) = \beta^H_{\mathsf{Val}} \circ (o.\mathit{fieldValue})$$

In particular for all fields, $f \in \mathrm{dom}(o.\mathit{fieldValue})$, it holds that $\beta^H_{\mathsf{Object}}(o)(f) = \beta^H_{\mathsf{Val}}(o.f)$.

The situation for arrays is slightly more involved since abstract arrays are represented as a single abstract value. It must therefore be ensured that all the elements of the concrete array are represented in this one abstract value which is done by simply taking the least upper bound of the abstract representation of all the elements of the concrete array. Let $a \in \mathsf{Array}$ and define:

$$\beta^H_{\mathsf{Array}}(a) = \bigsqcup_{0 \leq i \leq a.\mathit{length}} \beta^H_{\mathsf{Val}}(a.\mathit{values}(i))$$

Recall that the global heap is split into a static heap and an "ordinary heap". The static heap is used only for static fields and the ordinary heap contains both objects (with instance fields) and arrays.

Concrete heaps are defined as maps from locations to objects and arrays. In the analysis a location is abstracted into the class of object (type of array) that the location points to. There may therefore be several concrete objects (arrays) corresponding to a single abstract object (array) reference, i.e., an abstract reference may represent several different concrete locations. The abstract representation of a heap must therefore take all these concrete objects (arrays) into account; this is done by taking the least upper bound over all the involved objects (arrays):

$$\beta_{\mathsf{Heap}}(H)(\mathsf{Ref}\ \sigma) = \bigsqcup_{\substack{\mathit{loc} \in \mathrm{dom}(H) \\ \beta^H_{\mathsf{Val}}(\mathit{loc}) = (\mathsf{Ref}\ \sigma)}} \beta^H_{\mathsf{Object}}(H(\mathit{loc}))$$

and for array references

$$\beta_{\mathsf{Heap}}(H)(\mathsf{Ref}\ (\mathtt{array}\ t)) = \bigsqcup_{\substack{\mathit{loc} \in \mathrm{dom}(H) \\ \beta^H_{\mathsf{Val}}(\mathit{loc}) = (\mathsf{Ref}\ (\mathtt{array}\ t))}} \beta^H_{\mathsf{Array}}(H(\mathit{loc}))$$

Representing the static heap is much simpler as there are no references involved. The static heap just maps a field identifier to a value leading to a simple representation function for $K \in \mathsf{StaHeap}$:

$$\beta^H_{\mathsf{StaHeap}}(K) = \beta^H_{\mathsf{Val}} \circ K$$

which is very similar to the representation function for local heaps.

The representation functions defined above formally relates the concrete semantic domains to their abstract counterparts. The next step then is to define when an analysis result can be said to correctly approximate the semantics. To

this end *correctness relations* are introduced that relate analysis results to stack frames, call stacks, and semantic configurations. Correctness relations are used instead of representation functions because neither stack frames (and thus call stacks) nor semantic configurations have direct abstract counterparts.

Intuitively an abstract local heap and an abstract stack correctly approximates a stack frame if the abstract representation of the concrete local heap and stack are contained in their abstract equivalents (at the right address). In symbols:

$$\langle m, pc, L, S\rangle \; \mathcal{R}^H_{\mathsf{Frame}} \; (\hat{L}, \hat{S}) \quad \text{iff} \quad \begin{array}{l} \beta^H_{\mathsf{LocHeap}}(L) \sqsubseteq \hat{L}(m, pc) \wedge \\ \beta^H_{\mathsf{Stack}}(S) \sqsubseteq \hat{S}(m, pc) \end{array}$$

Note that correctness relations, like representation functions, may be defined relative to a heap, $H$.

Since call-stacks are finite sequences of stack frames, $SF = F_1 :: \cdots :: F_n$ where $F_i = \langle m_i, pc_i, L_i, S_i\rangle$ for $1 \leq i \leq n$, it is trivial to extend the correctness relation for stack frames to cover entire call-stacks by simply requiring that each individual stack frame is correctly represented:

$$SF \; \mathcal{R}^H_{\mathsf{CallStack}} \; (\hat{L}, \hat{S}) \quad \text{iff} \quad \forall i \in \{1, \ldots, n\} \colon F_i \; \mathcal{R}^H_{\mathsf{Frame}} \; (\hat{L}, \hat{S})$$

Finally, by combining all of the above representation functions and correctness relations, the correctness relation for full semantic configurations is defined as follows:

$$\langle K, H, SF\rangle \; \mathcal{R}_{\mathsf{Conf}} \; (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \quad \text{iff} \quad \begin{array}{l} \beta^H_{\mathsf{StaHeap}}(K) \sqsubseteq \hat{K} \wedge \\ \beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \wedge \\ SF \; \mathcal{R}^H_{\mathsf{CallStack}} \; (\hat{L}, \hat{S}) \end{array}$$

The above formalises the intuition that an analysis result correctly approximates a semantic configuration if the abstract representations, as defined by the representation functions, of the semantic objects that constitute the configuration are contained in the analysis result.

### Subject Reduction

Implicit in the analysis is an assumption that the structure of the call stacks is the result of method invocations and their corresponding returns. In particular it is assumed that all stack frames below the currently active stack frame are suspended in a method invocation waiting to return. A call-stack that conforms to this structure is called *well-formed*:

**Definition 3.10 (Well-Formedness).** *A call stack* $SF = \langle m_1, pc_1, L_1, S_1\rangle ::$

$\cdots :: \langle m_n, pc_n, L_n, S_n \rangle$, *is said to be* well-formed *if and only if*

$$
\begin{array}{l}
\forall i \in \{2, \ldots, n\} : \\
\quad m_i.instructionAt(pc_i) = \texttt{invokestatic}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: S_i'' \wedge \\
\qquad m_{i-1} = m_i'' \\
\quad m_i.instructionAt(pc_i) = \texttt{invokevirtual}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: loc_i :: S_i'' \wedge \\
\qquad H(loc_i).class = \sigma_i \wedge \\
\qquad m_{i-1} = methodLookup(m_i'', \sigma_i) \\
\quad m_i.instructionAt(pc_i) = \texttt{invokeinterface}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: loc_i :: S_i'' \wedge \\
\qquad H(loc_i).class = \sigma_i \wedge \\
\qquad m_{i-1} = methodLookup(m_i'', \sigma_i)
\end{array}
$$

This definition is extended to semantic configurations: a semantic configuration $\langle K, H, SF \rangle$ is *well-formed* if and only if $SF$ is well-formed.

Next two lemmas are proved showing that well-formedness of semantic configurations is preserved under semantic reduction and that initial configurations are always well-formed:

**Lemma 3.11.** *For $P \in$ Program if $\langle K, H, SF \rangle$ is a well-formed configuration and $P \vdash \langle K, H, SF \rangle \Longrightarrow \langle K', H', SF' \rangle$ then $\langle K', H', SF' \rangle$ is well-formed.*

*Proof.* By induction in the length of the call-stack, $SF$.

The base case, for call stacks of length one, holds vacuously. For the induction step, a case analysis is used on the instruction that is executed in the semantic step. There are only four interesting cases, since most instructions only modify the top element of the call stack. The `return` instruction on the other hand, removes the top element of the call-stack thus *reducing* the size of the call-stack and the case then follows immediately from the induction hypothesis. The remaining four cases (`invokevirtual`, `invokeinterface`, and `invokestatic`) follow from inspection of the semantics. ∎

**Lemma 3.12.** *If $\langle K, H, SF \rangle$ is an initial configuration for $P$ then $\langle K, H, SF \rangle$ is well-formed.*

*Proof.* Holds Vacuously since $|SF| = 1$. ∎

These two lemmas in conjunction justify the assumption made in the analysis and ensures that it is not necessary to consider any pathological call-stacks when proving the subject reduction property.

An immediate corollary of the two lemmas is that, for a given program, any semantic reduction sequence will consist entirely of well-formed configurations:

**Corollary 3.13.** *If $P \in$ Program, $C_0$ is an initial configuration for $P$, and $P \vdash C_0 \Longrightarrow^* C$ then $C$ is well-formed.*

The subject reduction property for the analysis can now be stated and proved:

**Theorem 3.14 (Subject Reduction).** *Let* $P \in \mathsf{Program}$, $(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{CFA} P$ *and let* $C = \langle K, H, SF\rangle$ *be a well-formed semantic configuration such that* $P \vdash C \Longrightarrow C'$ *then*

$$C\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \Rightarrow C'\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S})$$

*Proof.* By case inspection using Lemma 3.11 for the `return` instruction. Below a few illustrative cases are shown in detail.

*Case* `push`*:* By assumption:

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ t\ \ c}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, c :: S\rangle :: SF\rangle}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{push}\ \ t\ \ c \tag{3.1}$$

and

$$\langle K, H, \langle m, pc, V, S\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.2}$$

Now it follows from (3.1) that

$$\{\mathsf{INT}\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1) \tag{3.3}$$
$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \tag{3.4}$$

and from (3.2) it follows that

$$\beta^H(K) \sqsubseteq \hat{K} \tag{3.5}$$
$$\beta(H) \sqsubseteq \hat{H} \tag{3.6}$$
$$\beta^H(L) \sqsubseteq \hat{L}(m, pc) \tag{3.7}$$
$$\beta^H(S) \sqsubseteq \hat{S}(m, pc) \tag{3.8}$$

From (3.3) and (3.8) above it follows that

$$\begin{array}{rcl} \beta^H(c :: S) & = & \beta^H(c) :: \beta^H(S) \\ & = & \{\mathsf{INT}\} :: \beta^H(S) \\ & \sqsubseteq & \{\mathsf{INT}\} :: \hat{S}(m, pc) \\ & \sqsubseteq & \hat{S}(m, pc+1) \end{array} \tag{3.9}$$

By (3.4) and (3.7) the following is obtained:

$$\begin{array}{rcl} \beta^H(L) & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc+1) \end{array} \tag{3.10}$$

The Theorem now follows from (3.5), (3.6), (3.9) and (3.10).

*Case* `store`*:* By assumption:

$$\frac{m.instructionAt(pc) = \texttt{store}\ t\ x}{\begin{array}{l}P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle K, H, \langle m, pc+1, L[x \mapsto v], S\rangle :: SF\rangle\end{array}}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{store}\ t\ x \tag{3.11}$$

and

$$\langle K, H, \langle m, pc, L, v :: S\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.12}$$

From (3.12) it follows that

$$\begin{aligned} \beta^H(K) &\sqsubseteq \hat{K} && (3.13)\\ \beta(H) &\sqsubseteq \hat{H} && (3.14)\\ \beta^H(L) &\sqsubseteq \hat{L}(m, pc) && (3.15)\\ \beta^H(v :: S) &\sqsubseteq \hat{S}(m, pc) && (3.16)\end{aligned}$$

Now $\hat{S}(m, pc) = A :: X$ for some $A$ and $X$, and (3.16) implies that

$$\begin{aligned} \beta^H(v) &\sqsubseteq A && (3.17)\\ \beta^H(S) &\sqsubseteq X && (3.18)\end{aligned}$$

Using (3.11) gives

$$\begin{aligned} A &\sqsubseteq \hat{L}(m, pc+1)(x) && (3.19)\\ X &\sqsubseteq \hat{S}(m, pc+1) && (3.20)\end{aligned}$$

Combining (3.17) and (3.19) the following is obtained:

$$\begin{aligned} \beta^H(v) &\sqsubseteq A \\ &\sqsubseteq \hat{L}(m, pc+1)(x)\end{aligned} \tag{3.21}$$

and combining (3.18) and (3.20) it follows that

$$\begin{aligned} \beta^H(S) &\sqsubseteq X \\ &\sqsubseteq \hat{S}(m, pc+1)\end{aligned} \tag{3.22}$$

From (3.11) and the definition of $\sqsubseteq_x$ it follows that

$$\forall x \in \text{dom}(\hat{L}(m, pc)) \setminus \{x\} : \hat{L}(m, pc)(x) \sqsubseteq \hat{L}(m, pc+1)(x) \tag{3.23}$$

Thus from (3.21) and (3.23):

$$\beta^H(L[x \mapsto v]) \sqsubseteq \hat{L}(m, pc+1) \tag{3.24}$$

The Theorem now follows from (3.13), (3.14), (3.22) and (3.24).

*Case* `putfield`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield } f \\ f.isStatic = false \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ o' = o[fieldValue \mapsto o.fieldValue[f \mapsto v]] \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle m, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle K, H[loc \mapsto o'], \langle m, pc+1, L, S\rangle :: SF\rangle \end{array}}$$

Again by assumption:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{putfield } f \tag{3.25}$$

and

$$\langle K, H, \langle m, pc, L, v :: loc :: S\rangle :: SF\rangle \; \mathcal{R}_{\mathsf{Conf}} \; (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.26}$$

From (3.26) it follows that

$$\beta^H(K) \sqsubseteq \hat{K} \tag{3.27}$$
$$\beta(H) \sqsubseteq \hat{H} \tag{3.28}$$
$$\beta^H(V) \sqsubseteq \hat{L}(m, pc) \tag{3.29}$$
$$\beta^H(v :: loc :: S) \sqsubseteq \hat{S}(m, pc) \tag{3.30}$$

Clearly, $o.class = o'.class$, whence $H(loc).class = H'(loc).class$. This implies that

$$\beta^H_{\mathsf{Ref}}(loc) = \beta^{H'}_{\mathsf{Ref}}(loc) \tag{3.31}$$

and therefore

$$\beta^H_{\mathsf{Val}} = \beta^{H'}_{\mathsf{Val}} \tag{3.32}$$

whence

$$\beta^H_{\mathsf{Stack}} = \beta^{H'}_{\mathsf{Stack}} \tag{3.33}$$
$$\beta^H_{\mathsf{LocHeap}} = \beta^{H'}_{\mathsf{LocHeap}} \tag{3.34}$$
$$\beta^H_{\mathsf{StaHeap}} = \beta^{H'}_{\mathsf{StaHeap}} \tag{3.35}$$
$$\beta^H_{\mathsf{Object}} = \beta^{H'}_{\mathsf{Object}} \tag{3.36}$$

Now it follows from (3.27) and (3.35) that

$$\begin{array}{rcl} \beta^{H'}(K) & = & \beta^H(K) \\ & \sqsubseteq & \hat{K} \end{array} \tag{3.37}$$

And from (3.25), (3.29) and (3.34) it follows that

$$\begin{array}{rcl} \beta^{H'}(L) & = & \beta^H(L) \\ & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc+1) \end{array} \tag{3.38}$$

From (3.25) it is known that $\hat{S}(m, pc) = A :: B :: X$ for some $A$, $B$, and $X$ and thus from (3.30):

$$\beta^H(v) \sqsubseteq A \tag{3.39}$$
$$\beta^H(loc) \sqsubseteq B \tag{3.40}$$
$$\beta^H(S) \sqsubseteq X \tag{3.41}$$

Thus from (3.25), (3.33), and (3.41):

$$\begin{array}{rcl} \beta^{H'}(S) & = & \beta^H(S) \\ & \sqsubseteq & X \\ & \sqsubseteq & \hat{S}(m, pc+1) \end{array} \tag{3.42}$$

Let $\sigma = o.class$, then $\beta^H(loc) = \beta^{H'}(loc) = (\mathsf{Ref}\ \sigma)$. Since $H' = H[loc \mapsto o']$ it is the case that $\forall(\mathsf{Ref}\ \sigma')$ such that $(\mathsf{Ref}\ \sigma) \neq (\mathsf{Ref}\ \sigma')$:

$$\begin{array}{rcl} \beta(H')(\mathsf{Ref}\ \sigma') & = & \beta(H)(\mathsf{Ref}\ \sigma') \\ & \sqsubseteq & \hat{H}(\mathsf{Ref}\ \sigma') \end{array} \tag{3.43}$$

Similarly, for all $f'$ such that $f' \neq f$ it is the case that

$$\begin{array}{rcl} \beta(H')(\mathsf{Ref}\ \sigma)(f') & = & \beta(H)(\mathsf{Ref}\ \sigma)(f') \\ & \sqsubseteq & \hat{H}(\mathsf{Ref}\ \sigma)(f') \end{array} \tag{3.44}$$

The following is derived from (3.31) and (3.40):

$$\begin{array}{rcl} \beta^{H'}(loc) & = & \beta^H(loc) \\ & \sqsubseteq & B \end{array} \tag{3.45}$$

Putting (3.25), (3.32), (3.39), and (3.45) together results in

$$\begin{array}{rcl} \beta^{H'}(H'(loc).f) & = & \beta^{H'}(o'.f) \\ & = & \beta^{H'}(v) \\ & = & \beta^H(v) \\ & \sqsubseteq & A \\ & \sqsubseteq & \hat{S}(\mathsf{Ref}\ \sigma)(f) \end{array} \tag{3.46}$$

Whence, by (3.43), (3.44), and (3.46):

$$\beta(H') \sqsubseteq \hat{H} \tag{3.47}$$

The Theorem now follows from (3.37), (3.38), (3.42), and (3.47).

*Case* `getfield this`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield this}\ f \\ f.isStatic = false \qquad loc = L(0) \qquad loc \neq \texttt{null} \\ o = H(loc) \qquad v = o.fieldValue(f) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

Furthermore, by assumption:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{getfield this } f \tag{3.48}$$

and

$$\langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \; \mathcal{R}_{\mathsf{Conf}} \; (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.49}$$

From (3.49) it follows that

$$\begin{aligned} \beta^H(K) &\sqsubseteq \hat{K} && (3.50)\\ \beta(H) &\sqsubseteq \hat{H} && (3.51)\\ \beta^H(L) &\sqsubseteq \hat{L}(m, pc) && (3.52)\\ \beta^H(S) &\sqsubseteq \hat{S}(m, pc) && (3.53) \end{aligned}$$

It follows directly from (3.48) and (3.52) that

$$\begin{aligned} \beta^H(L) &\sqsubseteq \hat{L}(m, pc)\\ &\sqsubseteq \hat{L}(m, pc+1) \end{aligned} \tag{3.54}$$

Now assuming that $o.class = \sigma$ it follows from (3.52) that

$$\begin{aligned} \beta^H(L(0)) &= \beta^H(loc)\\ &= (\mathsf{Ref}\ \sigma)\\ &\sqsubseteq \hat{L}(m, pc)(0) \end{aligned} \tag{3.55}$$

Then from (3.48) and (3.55):

$$\hat{H}(\mathsf{Ref}\ \sigma)(f) :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1) \tag{3.56}$$

Using (3.51), the fact that $\sigma = o.class = H(loc).class$ and the definition of $\beta(H)(\mathsf{Ref}\ \sigma)$ results in

$$\begin{aligned} \beta^H(H(loc)) &= \beta^H(o)\\ &\sqsubseteq \beta(H)(\mathsf{Ref}\ \sigma)\\ &\sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma) \end{aligned} \tag{3.57}$$

and thus

$$\beta^H(o.fieldValue(f)) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f) \tag{3.58}$$

and therefore

$$\begin{aligned} \beta^H(v :: S) &= \beta^H(v) :: \beta^H(S)\\ &\sqsubseteq \beta^H(v) :: \hat{S}(m, pc)\\ &= \beta^H(o.fieldValue(f)) :: \hat{S}(m, pc)\\ &\sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f) :: \hat{S}(m, pc)\\ &\sqsubseteq \hat{S}(m, pc+1) \end{aligned} \tag{3.59}$$

The Theorem now follows from (3.50), (3.51), (3.54) and (3.59).

*Case* `invokevirtual`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ \ m_0 \\ m.isStatic = false \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ L_v = loc :: v_1 \cdots :: v_{|m_0|} m_v = methodLookup(m_0, o.class) \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \quad \langle K, H, \langle m_v, 0, L_v, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \end{array}}$$

and furthermore

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual}\ \ m_0 \tag{3.60}$$

and

$$\langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle)\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.61}$$

From (3.61) it immediately follows that

$$\beta^H(K) \sqsubseteq \hat{K} \tag{3.62}$$
$$\beta(H) \sqsubseteq \hat{H} \tag{3.63}$$
$$\beta^H(L) \sqsubseteq \hat{L}(m, pc) \tag{3.64}$$
$$\beta^H(v_1 :: \cdots :: v_n :: loc :: S) \sqsubseteq \hat{S}(m, pc) \tag{3.65}$$

From (3.60) it follows that $\hat{S}(m, pc) = A_1 :: \cdots :: A_{|m_0|} :: B :: X$ for some $A_1, \ldots, A_{|m_0|}, B, X$ and thus (3.61) entails that

$$\beta^H(v_i) \sqsubseteq A_i \tag{3.66}$$
$$\beta^H(loc) \sqsubseteq B \tag{3.67}$$
$$\beta^H(S) \sqsubseteq X \tag{3.68}$$

Now, assuming that $o.class = \sigma$ it follows that $\beta^H(loc) = (\mathsf{Ref}\ \sigma)$ and combining (3.60) with (3.67) it gives rise to:

$$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|] \tag{3.69}$$

because $m_v = methodLookup(m_0, \sigma)$. From this it is clear that

$$\beta^H(L_v) \sqsubseteq \hat{L}(m_v, 0) \tag{3.70}$$

and

$$\beta^H(\epsilon) \sqsubseteq \hat{S}(m_v, 0) \tag{3.71}$$

The Theorem now follows from (3.62), (3.63), (3.70) and (3.71).

*Case* `return`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{return}\ \ t \\ S' = \begin{cases} v_1' :: \cdots :: v_{|m|}' :: loc :: S'' & \text{if } m.isStatic \neq true \\ v_1' :: \cdots :: v_{|m|}' :: S'' & \text{if } m.isStatic = true \end{cases} \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v :: S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle m', pc' + 1, L', v :: S''\rangle :: SF\rangle \end{array}}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{return}\ t \tag{3.72}$$

and

$$\langle K, H, \langle m, pc, L, v :: S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.73}$$

From the assumptions it follows immediately that

$$\begin{aligned}
\beta^H(K) &\sqsubseteq \hat{K} && (3.74)\\
\beta(H) &\sqsubseteq \hat{H} && (3.75)\\
\beta^H(L) &\sqsubseteq \hat{L}(m, pc) && (3.76)\\
\beta^H(v :: S) &\sqsubseteq \hat{S}(m, pc) && (3.77)\\
\beta^H(L') &\sqsubseteq \hat{L}(m', pc') && (3.78)\\
\beta^H(S') &\sqsubseteq \hat{S}(m', pc') && (3.79)
\end{aligned}$$

From Lemma 3.11 it follows that the instruction at $m'.instructionAt(pc')$ is one of either `invokevirtual`, `invokestatic` or `invokespecial` and thus:

$$\hat{L}(m', pc') \sqsubseteq \hat{L}(m', pc' + 1) \tag{3.80}$$

Combining this with (3.78) results in

$$\beta^H(L') \sqsubseteq \hat{L}(m', pc' + 1) \tag{3.81}$$

There are now three sub-cases to consider, depending on the instruction at $m'.instructionAt(pc')$. Assume now that

$$m'.instructionAt(pc') = \texttt{invokevirtual}\ m''$$

then it must be the case that $m.isStatic \neq true$ and then Lemma 3.11 gives that

$$m = methodLookup(m'', H(loc).class) \tag{3.82}$$

and

$$S' = v'_1 :: \cdots :: v_{|m|} :: loc :: S'' \tag{3.83}$$

From (3.79) it is the case that $\hat{S}(m', pc') = A_1 :: \cdots :: A_{|m|} :: B :: X$ and therefore that

$$H(loc).class = \beta^H(loc) \sqsubseteq B \tag{3.84}$$

Then by assumption

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m', pc') : \texttt{invokevirtual}\ m''$$

which combined with (3.82) and (3.84) gives that $\hat{S}(m, \mathsf{END}) = A :: Y$ and that

$$A :: X \sqsubseteq \hat{S}(m', pc' + 1) \tag{3.85}$$

but from (3.79) it is the case that

$$\beta^H(S'') \sqsubseteq X \tag{3.86}$$

and from (3.72) that $\hat{S}(m, pc) \sqsubseteq \hat{S}(m, \mathsf{END})$ which implies that

$$\beta^H(v :: S) = \beta^H(v) :: \beta^H(S) \sqsubseteq \hat{S}(m, \mathsf{END}) \tag{3.87}$$

and therefore it is known that

$$\beta^H(v) \sqsubseteq A \tag{3.88}$$

Combining the above results in the following

$$\begin{array}{rcl} \beta^H(v :: S'') & = & \beta^H(v) :: \beta^H(S'') \\ & \sqsubseteq & A :: X \\ & \sqsubseteq & \hat{S}(m', pc' + 1) \end{array} \tag{3.89}$$

The two remaining subcases are similar and the Theorem now follows from (3.74), (3.75), (3.81) and (3.89).

*Case* `arrayload`*:* By assumption:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{arrayload}\ t \\ loc \neq \texttt{null} \qquad v = H(loc).value(n) \qquad 0 \leq n < H(loc).length \end{array}}{\begin{array}{c} P \vdash \langle K, H, \langle m, pc, L, n :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle m, pc + 1, L, v :: S\rangle :: SF\rangle \end{array}}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{\text{CFA}} (m, pc) : \texttt{arrayload}\ t \tag{3.90}$$

and

$$\langle K, H, \langle m, pc, L, n :: loc :: S\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}) \tag{3.91}$$

and from this it immediately follows that

$$\beta^H(K) \sqsubseteq \hat{K} \tag{3.92}$$

$$\beta(H) \sqsubseteq \hat{H} \tag{3.93}$$

$$\beta^H(L) \sqsubseteq \hat{L}(m, pc) \tag{3.94}$$

$$\beta^H(n :: loc :: S) \sqsubseteq \hat{S}(m, pc) \tag{3.95}$$

Now it follows from (3.90) and (3.94) that

$$\begin{array}{rcl} \beta^H(L) & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc + 1) \end{array} \tag{3.96}$$

From (3.90) it follows that $\hat{S}(m, pc) = A :: B :: X$ for some $A$, $B$ and $X$; now (3.95) implies that

$$\beta^H(n) \sqsubseteq A \tag{3.97}$$
$$\beta^H(loc) \sqsubseteq B \tag{3.98}$$
$$\beta^H(S) \sqsubseteq X \tag{3.99}$$

Since $\beta^H(loc) = (\mathsf{Ref}\ (\mathtt{array}\ t))$, eq. (3.98) implies $(\mathsf{Ref}\ (\mathtt{array}\ t)) \in B$ and thus from (3.90) it can be deduced that

$$\hat{H}(\mathsf{Ref}\ (\mathtt{array}\ t)) :: X \sqsubseteq \hat{S}(m, pc+1) \tag{3.100}$$

Furthermore it is the case that

$$\begin{array}{rcl} \beta^H(v) & = & \beta^H(H(loc).value(n)) \\ & \sqsubseteq & \beta^H(H(loc)) \\ & \sqsubseteq & \beta(H)(\mathsf{Ref}\ (\mathtt{array}\ t)) \\ & \sqsubseteq & \hat{H}(\mathsf{Ref}\ (\mathtt{array}\ t)) \end{array} \tag{3.101}$$

Finally from (3.99), (3.101) and (3.90) it follows that

$$\begin{array}{rcl} \beta^H(v :: S) & = & \beta^H(v) :: \beta^H(S) \\ & \sqsubseteq & \hat{H}(\mathsf{Ref}\ (\mathtt{array}\ t)) :: X \\ & \sqsubseteq & \hat{S}(m, pc+1) \end{array} \tag{3.102}$$

The Theorem new follows from (3.92), (3.93), (3.96) and (3.102).

The remaining cases are similar. ■

It follows from the subject reduction property that the control flow analysis does indeed track the control flow of a program and in particular which object references are used for virtual method invocations, i.e., the object references that determine the outcome of a dynamic dispatch. In fact the above theorem combined with the definitions of the representation functions implies that the analysis not only tracks the control flow of a program but actually computes a sound approximation of the local heap and operand stack for *every instruction* in the program.

### 3.3.2 Moore Family Property

This section is devoted to establishing that the set of acceptable analyses, as specified by the control flow analysis, constitute a *Moore family*:

**Definition 3.15 (Moore family).** *Let $L = (L, \sqsubseteq_L)$ be a complete lattice, then a subset $S \subseteq L$ of $L$ is a* Moore family *if and only if it is closed under greatest lower bounds: $\forall S' \subseteq S : \sqcap S' \in S$.*

Trivially, a Moore family, $S$, contains both a least and a greatest element: $\sqcap S$ and $\sqcap \emptyset$ (which is equal to $\top_L$) respectively. Thus a Moore family is never empty.

**Theorem 3.16 (Moore family (CFA)).** *The set of acceptable analyses for a given program under $\models_{CFA}$ is a Moore family, i.e., for a program $P \in \mathsf{Program}$:*

$$\forall \mathcal{A}' \subseteq \{\mathcal{A} \mid \mathcal{A} \models_{CFA} P\} : \sqcap \mathcal{A}' \in \{\mathcal{A} \mid \mathcal{A} \models_{CFA} P\}$$

*Proof.* Let $P \in \mathsf{Program}$ and $\mathcal{A}_{i \in \mathcal{I}} \in \widehat{\mathsf{Analysis}}_{\mathsf{CFA}}$ for some index set $\mathcal{I}$ such that $\mathcal{A}_i \models_{\mathrm{CFA}} P$ for all $i \in \mathcal{I}$. Then $(\sqcap_{i \in I} A_i) \models_{\mathrm{CFA}} P$ can be proved by case inspection on the analysis specification for each instruction.

*Case* `push`*:* Let $(\hat{K}_i, \hat{H}_i, \hat{L}_i, \hat{S}_i) = \mathcal{A}_i$ for all $i \in \mathcal{I}$, then by definition $A_i \models_{\mathrm{CFA}} P$ is equivalent to

$$\begin{array}{l} \{\mathsf{INT}\} :: \hat{S}_i(m, pc) \sqsubseteq \hat{S}_i(m, pc+1) \\ \hat{L}_i(m, pc) \sqsubseteq \hat{L}_i(m, pc+1) \end{array}$$

and thus

$$\forall i \in \mathcal{I} : \{\mathsf{INT}\} \subseteq \hat{S}_i(m, pc+1)|_0 \Rightarrow \{\mathsf{INT}\} \subseteq (\sqcap_{i \in \mathcal{I}} \hat{S}_i)(m, pc+1)|_0$$

Similarly

$$\begin{array}{rcl} \forall j \in \mathcal{I} : \forall k \in \{0, \ldots, |\hat{S}_j|\} : (\sqcap_{i \in \mathcal{I}} \hat{S}_i)(m, pc)|_k & \subseteq & \hat{S}_j(m, pc)|_k \\ & \subseteq & \hat{S}_j(m, pc+1)|_{k+1} \end{array}$$

and thus

$$\begin{array}{l} \forall k \in \{0, \ldots, \min_{j \in \mathcal{I}}\{|\hat{S}_j|\}\} : \\ \quad (\sqcap_{i \in \mathcal{I}} \hat{S}_i)(m, pc)|_k \subseteq (\sqcap_{i \in \mathcal{I}} \hat{S}_i)(m, pc+1)|_{k+1} \end{array}$$

whence

$$\{\mathsf{INT}\} :: (\sqcap_{i \in \mathcal{I}} \hat{S}_i(m, pc)) \sqsubseteq (\sqcap_{i \in \mathcal{I}} \hat{S}_i(m, pc+1))$$

The formula $\hat{L}_i(m, pc) \sqsubseteq \hat{L}_i(m, pc)$ is equivalent to

$$\forall x \in \mathrm{dom}(\hat{L}_i(m, pc)) : \hat{L}(m, pc)(x) \subseteq \hat{L}_i(m, pc+1)(x)$$

and therefore

$$\begin{array}{l} \forall j \in \mathcal{I} : \forall x \in (\bigcap_{i \in \mathcal{I}} \mathrm{dom}(\hat{L}_i)) : \\ \quad \begin{array}{rcl} (\sqcap_{i \in \mathcal{I}} \hat{L}_i(m, pc))(x) & \subseteq & \hat{L}_j(m, pc)(x) \\ & \subseteq & \hat{L}_j(m, pc+1)(x) \end{array} \end{array}$$

which entails

$$\forall x \in (\bigcap_{i \in \mathcal{I}} \mathrm{dom}(\hat{L}_i)) : (\sqcap_{i \in \mathcal{I}} \hat{L}_i(m, pc))(x) \subseteq (\sqcap_{i \in \mathcal{I}} \hat{L}_i(m, pc+1))(x)$$

Combined with the above this results in

$$\begin{array}{l} \{\mathsf{INT}\} :: (\sqcap_{i \in \mathcal{I}} \hat{S}_i(m, pc)) \sqsubseteq (\sqcap_{i \in \mathcal{I}} \hat{S}_i(m, pc+1)) \\ (\sqcap_{i \in \mathcal{I}} \hat{L}_i(m, pc)) \sqsubseteq (\sqcap_{i \in \mathcal{I}} \hat{L}_i(m, pc+1)) \end{array}$$

which is equivalent to

$$(\sqcap_{i \in I} A_i) \models_{\mathrm{CFA}} (m, pc) : \texttt{push}\ t\ n$$

This ends the case for `push`.

The remaining cases are similar. ∎

From Theorem 3.16 and the comments following Definition 3.15 it follows that all programs admit at least one analysis, namely $\sqcap\emptyset$, and in addition that all programs admit a best (smallest) analysis, namely $\bigsqcap\{\mathcal{A} \mid \mathcal{A} \models_{\text{CFA}} P\}$.

## 3.4 Implementation

Having specified a control flow analysis and formally proved it correct, the focus is now turned to implementing the analysis. For analyses specified in the Flow Logic framework the most expedient way to do this is often by converting the abstract specification into a *constraint generator* over a suitable constraint language for which automated solvers exist. That is also the approach taken here by turning the specification into a constraint generator over the *Alternation-free Least Fixed-Point logic* (ALFP). Constraints in ALFP form can be solved efficiently by the "Succinct Solver" developed by Nielson and Seidl in [NS01]. A detailed discussion of ALFP and the "Succinct Solver" can be found in [NNS02b] while [NNS$^+$04] illustrates the solvers usefulness for program analysis applications.

### 3.4.1 Alternation-free Least Fixed-Point logic

The presentation of ALFP given here closely follows that given in [NNS02b].

Let $\mathcal{X}$ be a fixed countable set of variables and $\mathcal{R}$ a finite, ranked alphabet of predicate symbols. The preconditions, *pre*, and clauses, *cl*, of ALFP are then defined as follows:

$$\begin{array}{rcl} pre & ::= & R(x_1,\ldots,x_n) \mid \neg R(x_1,\ldots,x_n) \mid pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \\ & \mid & \exists x : pre \mid \forall x : pre \\ \\ cl & ::= & R(x_1,\ldots,x_n) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \end{array}$$

where $x, x_1, \ldots, x_n \in \mathcal{X}$ and $R \in \mathcal{R}$. Occurrences of $R(\cdots)$ and $\neg R(\cdots)$ in preconditions are called *queries* and *negative queries* respectively; occurrences in clauses are called *assertions*.

In order to handle negative queries a notion of *stratification* is introduced similar to that found in *Datalog*:

**Definition 3.17 (ALFP Formula).** *A clause, cl, is said to be an* ALFP formula *if it has the form* $cl = cl_1 \wedge \cdots \wedge cl_n$ *and there exists a* ranking function, $rank\colon \mathcal{R} \to \{1,\ldots,n\}$, *such that for* $i \in \{1,\ldots,n\}$ *the following hold:*

1. $rank(R) = i$ *for all predicates* $R$ *of an assertion in* $cl_i$
2. $rank(R) \leq i$ *for all predicates* $R$ *of a query in* $cl_i$
3. $rank(R) < i$ *for every predicate* $R$ *of a negative query in* $cl_i$

| | | |
|---|---|---|
| $(\rho,\sigma) \models_{\text{PRE}} R(x_1,\ldots,x_n)$ | iff | $(\sigma(x_1),\ldots,\sigma(x_n)) \in \rho(R)$ |
| $(\rho,\sigma) \models_{\text{PRE}} \neg R(x_1,\ldots,x_n)$ | iff | $(\sigma(t_1),\ldots,\sigma(t_n)) \notin \rho(R)$ |
| $(\rho,\sigma) \models_{\text{PRE}} pre_1 \wedge pre_2$ | iff | $(\rho,\sigma) \models_{\text{PRE}} pre_1$ and $(\rho,\sigma) \models_{\text{PRE}} pre_2$ |
| $(\rho,\sigma) \models_{\text{PRE}} pre_1 \vee pre_2$ | iff | $(\rho,\sigma) \models_{\text{PRE}} pre_1$ or $(\rho,\sigma) \models_{\text{PRE}} pre_2$ |
| $(\rho,\sigma) \models_{\text{PRE}} \exists x : pre$ | iff | $(\rho,\sigma[x \mapsto a]) \models_{\text{PRE}} pre$ for some $a \in \mathcal{U}$ |
| $(\rho,\sigma) \models_{\text{PRE}} \forall x : pre$ | iff | $(\rho,\sigma[x \mapsto a]) \models_{\text{PRE}} pre$ for all $a \in \mathcal{U}$ |
| $(\rho,\sigma) \models_{\text{ALFP}} R(x_1,\ldots,x_n)$ | iff | $(\sigma(x_1),\ldots,\sigma(x_n)) \in \rho(R)$ |
| $(\rho,\sigma) \models_{\text{ALFP}} \mathbf{1}$ | iff | *true* |
| $(\rho,\sigma) \models_{\text{ALFP}} cl_1 \wedge cl_2$ | iff | $(\rho,\sigma) \models_{\text{ALFP}} cl_1$ and $(\rho,\sigma) \models_{\text{ALFP}} cl_2$ |
| $(\rho,\sigma) \models_{\text{ALFP}} pre \Rightarrow cl$ | iff | $(\rho,\sigma) \models_{\text{ALFP}} cl$ whenever $(\rho,\sigma) \models_{\text{PRE}} pre$ |
| $(\rho,\sigma) \models_{\text{ALFP}} \forall x : cl$ | iff | $(\rho,\sigma[x \mapsto a]) \models_{\text{ALFP}} cl$ for all $a \in \mathcal{U}$ |

Figure 3.8: Semantics for ALFP

Each $i$ in the above definition is called a *stratum* and the maximal rank, i.e., the number of strata, is denoted by $\text{maxrank}(\mathcal{R}) = \max\{rank(R) \mid R \in \mathcal{R}\}$.

The semantics of clauses and preconditions can now be defined. Given a non-empty and finite universe, $\mathcal{U}$, of atomic values and interpretations $\rho$ and $\sigma$ for predicate symbols and free variables respectively, the satisfaction relations for preconditions and clauses are given in Figure 3.8.

For a given interpretation of free variables, $\sigma$, an interpretation, $\rho$, of predicate symbols is called a *solution* to the clause, $cl$, if indeed $(\rho,\sigma) \models_{\text{ALFP}} cl$. Furthermore, since the clauses considered in this dissertation have no free variables the given $\sigma$ is of no consequence and hence a fixed interpretation, $\sigma_0$, is assumed throughout this dissertation.

The set of interpretations of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$ gives rise to an ordering on the set of interpretations, $\Delta$. Let $\rho_1, \rho_2 \in \Delta$ and define

$$\begin{aligned}\rho_1 \sqsubseteq \rho_2 \quad \text{iff} \quad & \exists i \in \mathbb{N}_0: \\ & (\forall R \in \mathcal{R}\text{: } rank(R) < i \Rightarrow \rho_1(R) = \rho_2(R)) \wedge \\ & (\forall R \in \mathcal{R}\text{: } rank(R) = i \Rightarrow \rho_1(R) \subseteq \rho_2(R)) \wedge \\ & (i = \text{maxrank}(\mathcal{R}) \vee \\ & \quad \exists R \in \mathcal{R}\text{: } rank(R) = i \wedge \rho_1(R) \subset \rho_2(R))\end{aligned}$$

In fact $\Delta = (\Delta, \sqsubseteq)$ forms a complete lattice. Furthermore sets of solutions form Moore families, cf. Definition 3.15, as proved by Proposition 1 of [NNS02b]. This is an important property because it implies that there always exists a solution for a stratified ALFP clause with no free variables and indeed there always exists a *least* solution.

Using stratification it is possible to define equality and in-equality predicates in the following manner:

$$(\forall x : \mathsf{EQ}(x,x)) \wedge (\forall x : \forall y : \neg\mathsf{EQ}(x,y) \Rightarrow \mathsf{NEQ}(x,y))$$

To see that the above formula is properly stratified take $rank(\mathsf{EQ}) = 0$ and

$rank(\mathsf{NEQ}) = 1$. In the following the query $\mathsf{EQ}(x, y)$ is written $x = y$ and the query $\mathsf{NEQ}(x, y)$ as $x \neq y$.

### 3.4.2 Solving the Constraints

In [NNS02b, NS01] an algorithm for finding the least solution in an efficient manner is described along with the techniques used to implement an efficient prototype called the "Succinct Solver"[2]. The details of both algorithm and prototype implementation are outside the scope of this dissertation.

Various benchmarks and example uses of the Succinct Solver can be found in [NNS+04, BNN02, Pil03].

### 3.4.3 Generating Constraints

In the rest of this section a given finite universe, $\mathcal{U}_P$, is assumed; the universe is assumed to contain only the atoms that occur in the program $P \in \mathsf{Program}$. This includes all the abstract program structures such as classes and method and also the runtime structures such as objects, arrays and basic values.

Furthermore, in order to manipulate and calculate stack positions the universe is assumed to include a number of unique atoms that represent the stack height for a given program. This is possible since the maximum stack height of a program is known beforehand and is known to be finite. Assuming that the maximum stack height is $n$ then the stack position atoms are denoted: $[\![0]\!], \ldots, [\![n]\!]$. For the actual calculation of stack positions a number of auxiliary relations are also created for every possible stack move, e.g., the relation $\mathsf{SP}_{-2}$ corresponds to removing the top two stack positions, i.e., $\mathsf{SP}_{-2}(x, y)$ holds for all $x = [\![i]\!]$ and $y = [\![j]\!]$ such that $j = i - 2$. In general:

$$\forall i \in \{0, \ldots, n\} : i + j \geq 0 \Rightarrow \mathsf{SP}_j([\![i]\!], [\![i + j]\!])$$

Note that for a given program all the auxiliary relations that are needed in order to generate constraints can be determined statically since stack copying depends only depends on which instructions are used and the arity of methods that are invoked, e.g., either $\mathsf{SP}_{-|m_0|}$ (if $m_0$ returns a value) or $\mathsf{SP}_{-(|m_0|+1)}$ (if $m_0$ does not return a value) is needed for invoking method $m_0$. Thus copying a stack from address $(m, pc)$ forward to address $(m, pc + 1)$ less the top two positions can be formulated in ALFP as follows:

$$\forall x : \forall y : \forall v : \mathsf{SP}_{-2}(x, y) \wedge \mathsf{S}(m, pc, x, v) \Rightarrow \mathsf{S}(m, pc + 1, y, v)$$

The Succinct Solver actually has a built-in encoding of terms using relations. This allows for a simpler and more elegant encoding of stack positions using terms to encode *Peano numbers* that can then be manipulated directly without the need for (explicit) auxiliary relations.

[2] The name was derived from the fact that the algorithm can be described very succinctly in one page of Standard ML(-like) pseudo-code.

To enhance readability and ease the specification of the constraint generator two auxiliary predicates are introduced. One that copies the stack in method $m_0$ at program counter $pc_0$ from stack position $i$ (and down) to the stack at program counter $pc_1$ from stack position $j$:

$$\begin{array}{l}\mathsf{COPYSTACK}(m_0, pc_0, i, pc_1, j) \equiv \\ \quad \forall x : \forall y : \forall v : \mathsf{SP}_{j-i}(x, y) \wedge \mathsf{S}(m_0, pc_0, x, v) \Rightarrow \mathsf{S}(m_0, pc_1, y, v)\end{array}$$

The other predicate copies the local heap in method $m$ at program counter $pc_0$ to program counter $pc_1$:

$$\begin{array}{l}\mathsf{COPYLOCHEAP}(m_0, pc_0, pc_1) \equiv \\ \quad \forall x : \forall v : \mathsf{L}(m_0, pc_0, x, v) \Rightarrow \mathsf{L}(m_0, pc_1, x, v)\end{array}$$

Below the constraint generator is developed in detail for a few instructions. The specification for the rest of the instructions can be found in Appendix B.

### The `push` instruction

The `push` instruction places a number or a `null`-reference on top of the stack. Assuming that it is a number the Flow Logic specification would look like the following:

$$\{\mathsf{INT}\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$$

which is turned into two ALFP formulae, one that "pushes" the number on top of the stack:

$$\mathsf{S}(m, pc, [\![0]\!], \mathsf{INT})$$

and one that transfers the remainder of the stack:

$$\mathsf{COPYSTACK}(m, pc, 0, pc + 1, 1)$$

Furthermore, the local heap should be copied forward without modification. In the Flow Logic specification this is done by the following:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$

which is implemented by the COPYLOCHEAP predicate:

$$\mathsf{COPYLOCHEAP}(m, pc, pc + 1)$$

Putting all of the above together results in the following constraint generator for `push`:

$$\begin{array}{l}\mathcal{G}[\![(m, pc) : \texttt{push } t \ n]\!] = \\ \qquad \mathsf{S}(m, pc + 1, [\![0]\!], \mathsf{INT}) \wedge \\ \qquad \mathsf{COPYSTACK}(m, pc, 0, pc + 1, 1) \\ \qquad \mathsf{COPYLOCHEAP}(m, pc, pc + 1)\end{array}$$

### The `store` Instruction

The Flow Logic specification for the `store` instruction makes use of the stack matching and binding operator:

$$A :: X \triangleleft \hat{S}(m, pc)$$

Intuitively this operator binds whatever is on top of the stack to the variable $A$ which may then be referenced later in the specification. This is used to transfer the value on top of the stack to the local variable $x$:

$$A \sqsubseteq \hat{L}(m, pc+1)(x)$$

which is translated into the following ALFP formula:

$$\forall v : \mathsf{S}(m, pc, [\![0]\!], v) \;\Rightarrow\; \mathsf{L}(m, pc+1, x, v)$$

A more direct translation of the $A_0 :: \cdots :: A_n :: X \triangleleft \hat{S}(m, pc)$ notation would be to introduce fresh relations $\mathsf{A}_0, \ldots, \mathsf{A}_n$ and then copy the values from the stack to the new relations, e.g., $\forall v : \mathsf{S}(m, pc, [\![0]\!], v) \;\Rightarrow\; \mathsf{A}_0(v)$. From there the values could then be transferred to their final destination, e.g., $\forall v : \mathsf{A}_0(v) \;\Rightarrow\; \mathsf{L}(m, pc+1, x, v)$ as above. However, such a translation would introduce many new relations only used for temporarily storing values. The translation used here "short circuits" the process and thus obviates the need for creating new relations used only for temporary storage.

As for the `push` instruction another formula is needed to copy the remainder of the stack:

$$\mathsf{COPYSTACK}(m, pc, 1, pc+1, 0)$$

Finally the local heap is copied forward while taking care to model the special operator $\sqsubseteq_X$ that transfers the contents of all the variables in the local heap except those in $X$:

$$\hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1)$$

this functionality is implemented by comparing the variable names for inequality using the encoded predicate $t_1 \neq t_2$:

$$\forall y : \forall v : (y \neq x) \wedge \mathsf{L}(m, pc, y, v) \;\Rightarrow\; \mathsf{L}(m, pc+1, y, v)$$

Thus the constraint generator for `store`:

$$\begin{aligned}
&\mathcal{G}[\![(m, pc) : \texttt{store } t\ x]\!] \;=\; \\
&\qquad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \;\Rightarrow\; \mathsf{L}(m, pc+1, x, v) \wedge \\
&\qquad \forall y : \forall v : (y \neq x) \wedge \mathsf{L}(m, pc, y, v) \;\Rightarrow\; \mathsf{L}(m, pc+1, y, v) \\
&\qquad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 0)
\end{aligned}$$

### The `invokevirtual` Instruction

The constraint generator for `invokevirtual` is the most involved simply because of its size. For ease of reference the Flow Logic specification is repeated here:

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S}) \models_{\text{CFA}} (m,pc) : \texttt{invokevirtual}\ \ m_0 \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \forall(\mathsf{Ref}\ \sigma) \in B : \\
\qquad\quad m_v \triangleleft \mathit{methodLookup}(m_0,\sigma) : \\
\qquad\quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v,0)[0..|m_0|] \\
\qquad\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad A :: Y \triangleleft \hat{S}(m_v,\mathsf{END}) :\ A :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)
\end{array}
$$

First the target method is looked up using the encoding of the *methodLookup* function and the object reference found on the stack. The reference on the stack is then transferred to the invoked method's local variable 0 as a self-reference:

$$\forall r : \forall m_v : \mathsf{S}(m,pc,[\![|m_0|]\!],r) \wedge \mathsf{ML}(m,r,m_v) \Rightarrow \mathsf{L}(m_v,0,0,r)$$

Next the parameters are transferred in much the same way, with one formula for each parameter:

$$
\begin{array}{l}
\forall r : \forall m_v : \forall v : \\
\quad \mathsf{S}(m,pc,[\![|m_0|]\!],r) \wedge \mathsf{ML}(m,r,m_v) \wedge \\
\qquad \mathsf{S}(m,pc,[\![0]\!],v) \Rightarrow \mathsf{L}(m_v,0,1,v) \\
\qquad\qquad\qquad \vdots \\
\forall r : \forall m_v : \forall v : \\
\quad \mathsf{S}(m,pc,[\![|m_0|]\!],r) \wedge \mathsf{ML}(m,r,m_v) \wedge \\
\qquad \mathsf{S}(m,pc,[\![|m_0|-1]\!],v) \Rightarrow \mathsf{L}(m_v,0,|m_0|,v)
\end{array}
$$

Since no local variables were modified in the invoking method, the local heap is simply copied:

$$\mathsf{COPYLOCHEAP}(m,pc,pc+1)$$

If the invoked method does not return a value, i.e., if $m_0.\mathit{returnType} = \texttt{void}$ the remainder of the stack is copied forward:

$$\mathsf{COPYSTACK}(m,pc,|m_0|+1,pc+1,0)$$

If instead the method does return a value, it should be copied from the top of the invoked method's stack and back to the stack of the invoking method:

$$
\begin{array}{l}
\forall r : \forall m_v : \forall v : \\
\quad \mathsf{S}(m,pc,|m_0|,r) \wedge \mathsf{ML}(m,r,m_v) \wedge \\
\qquad \mathsf{S}(m_v,\mathsf{END},[\![0]\!],v) \Rightarrow \mathsf{S}(m,pc+1,[\![0]\!],v)
\end{array}
$$

and the remaining stack is copied:

$$\mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 1)$$

Combining the above results in the following constraint generator:

$$\begin{array}{l}
\mathcal{G}[\![(m, pc) : \texttt{invokevirtual } m_0]\!] = \\
\quad \forall r : \forall m_v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \Rightarrow \mathsf{L}(m_v, 0, 0, r) \\
\quad \forall r : \forall m_v : \forall v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m_v, 0, 1, v) \\
\quad\quad \vdots \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![|m_0| - 1]\!], v) \Rightarrow \mathsf{L}(m_v, 0, |m_0|, v) \\
\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1) \\
\quad \text{if } m_0.returnType = \texttt{void} \text{ then} \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 0) \\
\quad \text{if } m_0.returnType \neq \texttt{void} \text{ then} \\
\quad\quad \forall r : \forall m_v : \forall v : \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m_v, \mathsf{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v) \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 1)
\end{array}$$

**Generating Constraints for Programs**

The constraint generators for the individual instructions are combined and lifted to covers entire programs:

$$\begin{array}{lll}
\mathcal{G}[\![P]\!] = & \bigwedge_{\substack{\forall (m, pc) \in P.addresses \\ m.instructionAt(pc) = \texttt{instr}}} & \mathcal{G}[\![(m, pc) : \texttt{instr}]\!] \\
\wedge & \bigwedge_{\substack{\sigma \in P.main \\ m_\sigma = \sigma.entry}} & \mathsf{L}(m_\sigma, 0, 0, (\mathsf{Ref}\ \sigma)) \\
\wedge & \bigwedge_{\substack{\sigma \in P.classes \\ m = \sigma.methods \\ m_v = methodLookup(m, \sigma)}} & \mathsf{ML}(m, \sigma, m_v)
\end{array}$$

where the ML relation is used to statically encode the *methodLookup* function to make it accessible in the ALFP formulation.

### 3.4.4 Correctness of the Constraint Generator

It is now possible to prove that the two formulations of the control flow analysis, as an abstract Flow Logic specification and as an ALFP constraint generator re-

spectively, are equivalent. First a lemma showing that the generated constraints do indeed form an ALFP formula:

**Lemma 3.18.** *Let* $P \in \mathsf{Program}$ *then* $\mathcal{G}[\![P]\!]$ *is an ALFP formula.*

*Proof.* Define $\mathit{rank}(\mathsf{EQ}) = 0$, $\mathit{rank}(\mathsf{NEQ}) = 1$, and let all other predicates have rank 2. It is easily checked that this ranking function fulfils the requirements of Definition 3.17 since negation is only used to define in-equality. ∎

In order to compare the ALFP solutions to solutions over the abstract domains a translation from the ALFP solution back to the abstract domains of the Flow Logic is defined:

$$\begin{array}{rcl}
[\![\mathsf{L}]\!]_\rho(m,pc)(x) & = & \{v \mid (m,pc,x,v) \in \rho(\mathsf{L})\} \\
[\![\mathsf{S}]\!]_\rho(m,pc)|_i & = & \{v \mid (m,pc,[\![i]\!],v) \in \rho(\mathsf{S})\} \\
[\![\mathsf{H}]\!]_\rho(\mathsf{Ref}\ \sigma)(f) & = & \{v \mid ((\mathsf{Ref}\ \sigma),f,v) \in \rho(\mathsf{H})\} \\
[\![\mathsf{H}]\!]_\rho(\mathsf{Ref}\ (\mathtt{array}\ t)) & = & \{v \mid ((\mathsf{Ref}\ (\mathtt{array}\ t)),\mathtt{ARRAY},v) \in \rho(\mathsf{H})\} \\
[\![\mathsf{K}]\!]_\rho(f) & = & \{v \mid (f,v) \in \rho(\mathsf{K})\}
\end{array}$$

which is combined to:

$$[\![\rho]\!] = ([\![\mathsf{K}]\!]_\rho, [\![\mathsf{H}]\!]_\rho, [\![\mathsf{L}]\!]_\rho, [\![\mathsf{S}]\!]_\rho)$$

And finally a lemma proving that the predicates introduced to conveniently copy stack positions and local heaps work as expected:

**Lemma 3.19.** *For any solution* $\rho$ *the following holds:*

$$\begin{array}{c}
(\rho,\sigma_0) \models_{\mathit{ALFP}} \mathsf{COPYSTACK}(m,pc_0,i,pc_1,j) \quad \textit{iff} \\
\forall x : [\![\mathsf{S}]\!]_\rho(m,pc_0)|_{x+i} \subseteq [\![\mathsf{S}]\!]_\rho(m,pc_1)|_{x+j}
\end{array}$$

*and*

$$\begin{array}{c}
(\rho,\sigma) \models_{\mathit{ALFP}} \mathsf{COPYLOCHEAP}(m,pc_0,pc_1) \quad \textit{iff} \\
\forall x : [\![\mathsf{L}]\!]_\rho(m,pc_0)(x) \subseteq [\![\mathsf{L}]\!]_\rho(m,pc_1)(x)
\end{array}$$

*Proof.* By expansion of the COPYSTACK and COPYLOCHEAP predicates and applying the semantics of ALFP. ∎

The equivalence between the two formulations of the analysis can now be stated and proved:

**Theorem 3.20.** *Let* $P \in \mathsf{Program}$ *then*

$$[\![\rho]\!] \models_{\mathit{CFA}} P \quad \textit{iff} \quad (\rho,\sigma_0) \models_{\mathit{ALFP}} \mathcal{G}[\![P]\!]$$

*Proof.* By case analysis on the instructions of $P$. Here only two cases are considered in detail. The remaining cases are similar or trivial.

*Case* `push`*: (if)* Assume that

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathcal{G}[\![(m, pc) : \texttt{push } t \ n]\!]$$

Using the definition of the constraint generator for `push` the above implies

$$\begin{aligned}(\rho, \sigma_0) \models_{\text{ALFP}} (&\mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \wedge \\ &\mathsf{COPYSTACK}(m, pc, 0, pc+1, 1) \wedge \\ &\mathsf{COPYLOCHEAP}(m, pc, pc+1))\end{aligned}$$

which is equivalent to

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \tag{3.103}$$
$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1) \tag{3.104}$$
$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathsf{COPYLOCHEAP}(m, pc, pc+1) \tag{3.105}$$

Applying the semantics of ALFP to equation (3.103) results in:

$$(m, pc, [\![0]\!], \mathsf{INT}) \in \rho(\mathsf{S})$$

and thus by definition of $[\![\mathsf{S}]\!]_\rho$:

$$\{\mathsf{INT}\} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc)|_0$$

Using lemma 3.19 on equations (3.104) and (3.105) implies

$$\forall x : [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+0} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc+1)|_{x+1}$$

and

$$\forall x : [\![\mathsf{L}]\!]_\rho(m, pc)(x) \subseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)(x)$$

Combining the above results in

$$\{\mathsf{INT}\} :: [\![\mathsf{S}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{S}]\!]_\rho(m, pc+1)$$

and

$$[\![\mathsf{L}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)$$

and thus

$$[\![\rho]\!] \models_{\text{CFA}} (m, pc) : \texttt{push } t \ n$$

*(only if)* Assume that

$$[\![\rho]\!] \models_{\text{CFA}} (m, pc) : \texttt{push } t \ n$$

which is equivalent to

$$\begin{aligned}&\{\mathsf{INT}\} :: [\![\mathsf{S}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{S}]\!]_\rho(m, pc+1) \\ &[\![\mathsf{L}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)\end{aligned}$$

Expanding the definition of $\sqsubseteq$ results in:

$$\begin{array}{l}\{\mathsf{INT}\} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc+1)|_0 \\ \forall x : [\![\mathsf{S}]\!]_\rho(m, pc)|_x \subseteq [\![\mathsf{S}]\!]_\rho(m, pc+1)|_{x+1} \\ \forall x : [\![\mathsf{L}]\!]_\rho(m, pc)(x) \subseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)(x)\end{array}$$

which is equivalent to

$$\begin{array}{l}\{\mathsf{INT}\} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc+1)|_0 \\ \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1) \\ \mathsf{COPYLOCHEAP}(m, pc, pc+1)\end{array}$$

and by definition

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathcal{G}[\![(m, pc) : \texttt{push}\ \ t\ \ n]\!]$$

Which concludes the case for `push`.

*Case* `invokevirtual`*: (if)* Assume that

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathcal{G}[\![(m, pc) : \texttt{invokevirtual}\ \ m_0]\!]$$

This expands to

$$\begin{array}{l}(\rho, \sigma_0) \models_{\text{ALFP}} \forall r : \forall m_v : \mathsf{S}(m, pc, |m_0|, r) \wedge \\ \quad \mathsf{ML}(m_0, r, m_v) \Rightarrow \mathsf{L}(m_v, 0, 0, r)\end{array} \tag{3.106}$$

and

$$\begin{array}{l}(\rho, \sigma_0) \models_{\text{ALFP}} \forall r : \forall m_v : \forall v : \\ \qquad \mathsf{S}(m, pc, |m_0|, r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\ \qquad\quad \mathsf{S}(m, pc, 0, v) \Rightarrow \mathsf{L}(m_v, 0, 1, v) \\ \qquad \vdots \\ \qquad \mathsf{S}(m, pc, |m_0|, r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\ \qquad\quad \mathsf{S}(m, pc, |m_0|-1, v) \Rightarrow \mathsf{L}(m_v, 0, |m_0|, v)\end{array} \tag{3.107}$$

and

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathsf{COPYLOCHEAP}(m, pc, pc+1) \tag{3.108}$$

and if $m_0.returnType = \texttt{void}$ then

$$\mathsf{COPYSTACK}(m, pc, |m_0|+1, pc+1, 0) \tag{3.109}$$

else if $m_0.returnType \neq \texttt{void}$ then

$$\begin{array}{l}\forall r : \forall m_v : \forall v : \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\ \quad \mathsf{S}(m_v, \texttt{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc+1, [\![0]\!], v) \\ \mathsf{COPYSTACK}(m, pc, |m_0|+1, pc+1, 1)\end{array} \tag{3.110}$$

Equation (3.106) can further be reformulated as:

$$\begin{array}{l}\forall r : \forall m_v : r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge \\ \quad \mathsf{ML}(m_0,r,m_v) \Rightarrow r \in [\![\mathsf{L}]\!]_\rho(m_v,0)(0)\end{array}$$

By definition $\mathsf{ML}(m_0,r,m_v)$ is equivalent to $m_v = \mathit{methodLookup}(m_0,r)$ and thus the above gives

$$\begin{array}{l}\forall r : r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge \\ \quad m_v = \mathit{methodLookup}(m_0,r) \Rightarrow r \in [\![\mathsf{L}]\!]_\rho(m_v,0)(0)\end{array}$$

Equation (3.107) can be reformulated in a similar manner:

$$\begin{array}{l}\forall r : \forall v : r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge m_v = \mathit{methodLookup}(m_0,r) \wedge \\ \quad v \in [\![\mathsf{S}]\!]_\rho(m,pc)|_0 \Rightarrow r \in [\![\mathsf{L}]\!]_\rho(m_v,0)(1) \\ \vdots \\ \forall r : \forall v : r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge m_v = \mathit{methodLookup}(m_0,r) \wedge \\ \quad v \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|-1} \Rightarrow r \in [\![\mathsf{L}]\!]_\rho(m_v,0)(|m_0|)\end{array}$$

which can be combined to the following:

$$\begin{array}{l}\forall i \in \{0,\ldots,|m_0|-1\} : \forall r : \forall v : \\ \quad r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge v \in [\![\mathsf{S}]\!]_\rho(m,pc)|_i \Rightarrow \\ \quad\quad v \in [\![\mathsf{L}]\!]_\rho(m_v,0)(i+1)\end{array}$$

Using the notation from the Flow Logic specification this can be written as:

$$\begin{array}{l}A_1 :: \cdots :: A_{|m_0|} :: B \triangleleft [\![\mathsf{S}]\!]_\rho(m,pc) : \\ \forall(\mathsf{Ref}\ \sigma) \in B : \\ \quad m_v = \mathit{methodLookup}(m_0,\sigma) \\ \quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq [\![\mathsf{L}]\!]_\rho(m_v,\sigma)[0..|m_0|]\end{array}$$

Lemma 3.19 and equation (3.108) gives

$$[\![\mathsf{L}]\!]_\rho(m,pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m,pc+1)$$

Expanding equation (3.109) results in

$$\begin{array}{l}m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\ \quad \forall x : [\![\mathsf{S}]\!]_\rho(m,pc)|_{x+|m_0|+1} \subseteq [\![\mathsf{S}]\!]_\rho(m,pc)|_{x+0}\end{array}$$

and similarly for equation (3.110):

$$\begin{array}{l}m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\ \quad \forall r : \forall v : r \in [\![\mathsf{S}]\!]_\rho(m,pc)|_{|m_0|} \wedge m_v = \mathit{methodLookup}(m_0,r) \wedge \\ \quad\quad v \in [\![\mathsf{S}]\!]_\rho(m_v,\mathsf{END})|_0 \Rightarrow v \in [\![\mathsf{S}]\!]_\rho(m,pc+1)|_0 \\ \quad \forall x : [\![\mathsf{S}]\!]_\rho(m,pc)|_{x+|m_0|+1} \subseteq [\![\mathsf{S}]\!]_\rho(m,pc)|_{x+1}\end{array}$$

Finally, putting all of the above together:

$$
\begin{array}{l}
A_1 :: \cdots :: A_{|m_0|} :: B \triangleleft [\![\mathsf{S}]\!]_\rho(m, pc) : \\
\forall(\mathsf{Ref}\ \sigma) \in B : \\
\quad m_v = \mathit{methodLookup}(m_0, \sigma) \\
\quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq [\![\mathsf{L}]\!]_\rho(m_v, \sigma)[0..|m_0|] \\
\quad [\![\mathsf{L}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m, pc+1) \\
\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\
\qquad \forall x : [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+|m_0|+1} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+0} \\
\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad \forall r : \forall v : r \in [\![\mathsf{S}]\!]_\rho(m, pc)|_{|m_0|} \wedge v \in [\![\mathsf{S}]\!]_\rho(m_v, \mathsf{END})|_0 \Rightarrow \\
\qquad\quad v \in [\![\mathsf{S}]\!]_\rho(m, pc+1)|_0 \\
\qquad \forall x : [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+|m_0|+1} \subseteq [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+1}
\end{array}
$$

and thus

$$[\![\rho]\!] \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual}\ \ m_0$$

*(only if)* Assume that

$$[\![\rho]\!] \models_{\text{CFA}} (m, pc) : \texttt{invokevirtual}\ \ m_0$$

By definition this is equivalent to

$$
\begin{array}{l}
A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft [\![\mathsf{S}]\!]_\rho(m, pc) : \\
\forall(\mathsf{Ref}\ \sigma) \in B : \\
\quad m_v = \mathit{methodLookup}(m_0, \sigma) \\
\quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq [\![\mathsf{L}]\!]_\rho(m_v, 0)[0..|m_0|] \\
\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\
\qquad X \sqsubseteq [\![\mathsf{S}]\!]_\rho(m, pc+1) \\
\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad A :: Y \triangleleft [\![\mathsf{S}]\!]_\rho(m_v, \mathsf{END}) :\ A :: X \sqsubseteq [\![\mathsf{S}]\!]_\rho(m, pc+1) \\
[\![\mathsf{L}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)
\end{array}
$$

which can be reformulated as

$$
\begin{array}{l}
\forall(\mathsf{Ref}\ \sigma) \in [\![\mathsf{S}]\!]_\rho(m, pc)|_{|m_0|} : \\
\quad m_v = \mathit{methodLookup}(m_0, \sigma) \\
\quad \{(\mathsf{Ref}\ \sigma)\} \subseteq [\![\mathsf{L}]\!](m_v, 0)(0) \\
\quad \forall v : v \in [\![\mathsf{S}]\!]_\rho(m, pc)|_0 \subseteq v \in [\![\mathsf{L}]\!](m_v, 0)(1) \\
\quad \vdots \\
\quad \forall v : v \in [\![\mathsf{S}]\!]_\rho(m, pc)|_{|m_0|-1} \subseteq v \in [\![\mathsf{L}]\!](m_v, 0)(|m_0|) \\
\quad \text{if } m_0.\mathit{returnType} = \texttt{void} \text{ then} \\
\qquad \forall x : \forall v : v \in [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+|m_0|+1} \Rightarrow v \in [\![\mathsf{S}]\!]_\rho(m, pc+1)|_{x+0} \\
\quad \text{if } m_0.\mathit{returnType} \neq \texttt{void} \text{ then} \\
\qquad \forall v : v \in [\![\mathsf{S}]\!]_\rho(m_v, \mathsf{END})|_0 \Rightarrow v \in [\![\mathsf{S}]\!]_\rho(m, pc+1)|_0 \\
\qquad \forall x : \forall v : v \in [\![\mathsf{S}]\!]_\rho(m, pc)|_{x+|m_0|+1} \Rightarrow v \in [\![\mathsf{S}]\!]_\rho(m, pc+1)|_{x+1} \\
[\![\mathsf{L}]\!]_\rho(m, pc) \sqsubseteq [\![\mathsf{L}]\!]_\rho(m, pc+1)
\end{array}
$$

Rewriting the above once more using the definition of $\mathsf{ML}$ and Lemma 3.19 gives:

$$\begin{array}{l}
\forall(\mathsf{Ref}\ \sigma) : \forall m_v : \mathsf{S}(m, pc, [\![|m_0|]\!], (\mathsf{Ref}\ \sigma)) \wedge \mathsf{ML}(m_0, \sigma, m_v) \Rightarrow \\
\quad \mathsf{L}(m_v, 0, 0, (\mathsf{Ref}\ \sigma)) \\
\quad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m_v, 0, 1, v) \\
\quad \vdots \\
\quad \forall v : \mathsf{S}(m, pc, [\![|m_0| - 1]\!], v) \Rightarrow \mathsf{L}(m_v, 0, |m_0|, v) \\
\quad \text{if } m_0.returnType = \texttt{void} \text{ then} \\
\qquad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 0) \\
\quad \text{if } m_0.returnType \neq \texttt{void} \text{ then} \\
\qquad \forall v : \mathsf{S}(m_v, \mathsf{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v) \\
\qquad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 1) \\
\mathsf{COPYLOCHEAP}(m, pc, pc + 1)
\end{array}$$

which is equivalent to

$$(\rho, \sigma_0) \models_{\text{ALFP}} \mathcal{G}[\![(m, pc) : \texttt{invokevirtual}\ m_0]\!]$$

This concludes the case for `invokevirtual`.

The remaining 30+ cases are similar. ∎

As a consequence of this result it is now possible to build a fully automated analysis tool for Carmel programs.

### 3.4.5 Complexity and Scalability

In [NNS02b] the runtime complexity of the algorithm implemented by the Succinct Solver is analysed and discussed at length. Proposition 2 in [NNS02b] shows that the asymptotic complexity of the solver algorithm is given by

$$\mathcal{O}(\#\rho + N^r \cdot n)$$

where $N$ is the size of the universe, $n$ the size of the constraint solved, $r$ the maximal nesting depth of quantifiers, and $\#\rho$ the sum of cardinalities of the predicates in the constraint.

For Carmel programs the size of the generated constraint is linear in the program size, $|P|$, and thus $n$ is bounded linearly by the program size. The universe is comprised by the program constants and structures and thus the size of the universe, $N$, is also linearly bounded by the program size. The maximal nesting depth is easily determined to be three by inspecting the constraint generator. Finally the cardinalities of the relations are bounded by

$$|\mathsf{Method}|_P \cdot |\mathsf{PC}|_P \cdot \max\{|\mathsf{Var}|_P, |\mathsf{Stack}|_P\} \cdot N$$

where $|\mathsf{Method}|_P$ is the number of methods in program $P$, $|\mathsf{PC}|_P$ the largest number of instructions in any method, $|\mathsf{Var}|_P$ the largest number of local variables used in any method, $|\mathsf{Stack}|_P$ the largest number of stack positions used

in any method, and $N$ is the size of the universe. All of the factors in the above formula are linearly bounded by the size of the program and thus the above is bounded by $\mathcal{O}(|P|^4)$. This results in the following upper bound for the asymptotic complexity of the solver algorithm for constraints generated from Carmel programs

$$\mathcal{O}(|P|^4 + |P|^3 \cdot |P|)$$

and thus a bound of $\mathcal{O}(|P|^4)$ is obtained.

In [Pil03, NNS+04] a prototype implementation of the analysis is benchmarked on a realistically sized electronic purse applet, called *Demoney*, that was developed as part of the SecSafe project (see [Siv03b, MM02, Mar02]). Using a naïve and unoptimised constraint generator, very similar to the one outlined in this section, it took on the order of 30 seconds to solve the constraints generated for Demoney *and* a partial implementation of the JCRE API. This gives a rough indication that the combination of Flow Logic and ALFP/Succinct Solver scales very well to applications of a realistic size.

## 3.5 Handling Exceptions

The control flow analysis specified and proved correct in the preceding sections was designed to cover the Carmel language with one notable caveat: the analysis did not include exceptions. This situation is rectified in the following where the control flow analysis is extended, in a very systematic and modular way, to cover not only full Carmel but also $\text{Carmel}_{\mathsf{EXC}}$ by integrating an *exception analysis* into the base analysis. The exception analysis is specified in such a way as to make it easy to include into other analyses especially those discussed in later sections and chapters.

While applications in general should be developed to be robust and able to cope with errors and exceptional situations, the often sensitive and critical nature of most smart card applications combined with the severely limited opportunities for user intervention and interaction makes good error handling *essential* for smart card applets. This requirement frequently means that errors and exceptions, where possible, must be handled gracefully within the application itself, as it is often impossible or impractical for the user to assist the application in recovering after an error. Thus only errors that cannot be handled internally or that require user notification should be reported to the user.

A rather different approach to exceptions in a high-assurance environment is taken in the SPARK Ada approach[Bar03] where programs are required to be *exception free*, i.e., it must be shown that a program cannot possibly give rise to an exception. Such a strict requirement makes good sense in certain high-assurance environments where it may not be possible, or even make sense, to (try to) recover from an error or exceptional situation. For typical smart card applications such an approach is excessive and it is often preferable to let the user or an operator know that an error occurred. However, an exception analysis, that over-approximates the exceptions thrown, can be used to verify that a given program can only give rise to a specific subset of exceptions or

even that the program does not give rise to any exceptions at all. In this way program analysis can be used to check a wide spectrum of exception policies.

### 3.5.1 Abstract Domains

As discussed in Chapter 2 exceptions are simply objects of a subclass of the `Throwable`-class and this naturally leads to the following abstract domain for exceptions:

$$\widehat{\mathsf{Exception}} = \overline{\mathsf{ObjRef}}$$

and the representation function for exceptions is identical to the representation function for references. Let $(\mathsf{Ref}\ \sigma_X) \in \widehat{\mathsf{Exception}}$, $H \in \mathsf{Heap}$, and define:

$$\beta^H_{\mathsf{Exc}}(\sigma_X) = \beta^H_{\mathsf{Ref}}(\mathsf{Ref}\ \sigma_X) \tag{3.111}$$

An *exception cache* is introduced to track exceptions that are not handled locally, i.e., in the method they are thrown:

$$\widehat{\mathsf{ExcCache}} = \mathsf{Method} \to \mathcal{P}(\widehat{\mathsf{Exception}})$$

Combining the exception cache with the domains for the control flow analysis the domain below for exception analyses is obtained:

$$\widehat{\mathsf{Analysis}}_{\mathsf{EXC}} = \widehat{\mathsf{StaHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocHeap}} \times \widehat{\mathsf{Stack}} \times \widehat{\mathsf{ExcCache}}$$

Exceptions that are handled in the method in which they are thrown are not explicitly tracked. Instead the semantics are modelled directly by copying the current local heap and a stack consisting only of the reference to the exception object to the program counter of the exception handler.

To simplify the presentation of the analysis specification the following *exception predicate* is defined that formalises the abstract exception handling:

$$\begin{array}{l}\mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X),(m,pc)) \equiv \\ \quad \mathit{findHandler}(m,pc,\sigma_X) = \bot\ \Rightarrow \\ \quad\quad (\mathsf{Ref}\ \sigma_X) \in \hat{E}(m) \\ \quad \mathit{findHandler}(m,pc,\sigma_X) = pc_X\ \Rightarrow \\ \quad\quad \{(\mathsf{Ref}\ \sigma_X)\} :: \epsilon \sqsubseteq \hat{S}(m,pc_X) \\ \quad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc_X)\end{array}$$

where *findHandler* is the same function that is used in the semantics. Intuitively the predicate checks if an exception of class $\sigma_X$, that was thrown at program counter $pc$ in method $m$, has a local handler by looking it up using the *findHandler* function. If there is an appropriate local handler the first program counter of the handler, $pc_X$, is returned and the operand stack and local heap is set up for the local handler. If a handler is not found locally, the exception is just recorded in the exception cache for the current method.

### 3.5.2 Flow Logic Specification for Exception Analysis

The judgements for the exception analysis are on the form:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\text{EXC}} (m, pc) : \texttt{instr}$$

where $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \in \widehat{\mathsf{Analysis}}_{\mathsf{EXC}}$. As for the control flow analysis the intuitive meaning of the above judgement is that $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E})$ is an acceptable, i.e., correct, analysis of the instruction `instr` found at address $(m, pc)$.

With an exception analysis component added to the control flow analysis it is now possible to extend the control flow analysis to cover the entire Carmel language. In particular, the `throw`-instruction can now be analysed. The `throw`-instruction is used to explicitly throw a programmer defined exception. The class of the exception to be thrown is given as an object reference on top of the operand stack. Using the HANDLE-predicate defined previously, the specification for this instruction is especially simple:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\text{EXC}} (m, pc) : \texttt{throw} \\ \quad \text{iff} \quad B :: X \triangleleft \hat{S}(m, pc) : \\ \qquad \forall(\mathsf{Ref}\ \sigma_X) \in B : \mathsf{HANDLE}_{(\hat{L}, \hat{S}, \hat{E})}((\mathsf{Ref}\ \sigma_X), (m, pc)) \\ \qquad \mathsf{HANDLE}_{(\hat{L}, \hat{S}, \hat{E})}((\mathsf{Ref}\ \texttt{NullPointerExc}), (m, pc))\end{array}$$

Note that throwing an exception may *itself* give rise to an exception if the exception reference is a `null`-reference. This shows how the runtime exceptions can be modelled in the analysis and thus how the control flow and exception analyses can extended to cover not only Carmel but all of $\text{Carmel}_{\mathsf{EXC}}$.

Next a few representative clauses for the exception analysis are given, showing how the judgements for the control flow analysis are easily adapted for the exception analysis. The modifications mainly consist in adding a HANDLE-predicate to the analysis of instructions that may throw a runtime exception:

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\text{EXC}} (m, pc) : \texttt{numop}\ \ t\ \ \mathit{binop}\ \ [t'] \\ \quad \text{iff} \quad A_1 :: A_2 :: X \triangleleft \hat{S}(m, pc) : \\ \qquad \{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m, pc + 1) \\ \qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1) \\ \qquad \mathsf{HANDLE}_{(\hat{L}, \hat{S}, \hat{E})}((\mathsf{Ref}\ \texttt{ArithmeticExc}), (m, pc))\end{array}$$

The necessary changes are similar for most of the remaining instructions. Only the instructions for method invocation (`invokevirtual`, `invokeinterface`, and `invokestatic`) are slightly more involved: in addition to tracking exceptions that may be thrown as the result of executing the instruction itself, it must also be ensured that any exceptions thrown and *not caught* by the invoked method are re-thrown in the current method. In the semantics this is handled directly by the two special reduction rules that apply for exception frames, cf. Figure 2.11.

The specification for `invokevirtual` follows:

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\mathrm{EXC}} (m,pc) : \texttt{invokevirtual}\ m_0 \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \forall(\mathsf{Ref}\ \sigma) \in B : \\
\qquad\quad m_v \triangleleft \mathit{methodLookup}(m_0,\sigma) : \\
\qquad\quad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v,0)[0..|m_0|] \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma_X) \in \hat{E}(m_v) \colon \mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X),(m,pc)) \\
\qquad\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad A :: Y \triangleleft \hat{S}(m_v,\mathsf{END}) :\ A :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\qquad \mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \texttt{NullPointerExc}),(m,pc))
\end{array}
$$

The analysis of the other instructions for method invocation are similar.

### 3.5.3 Semantic Correctness

Semantic correctness for the exception analysis is established in the same way as for the control flow analysis by proving a subject reduction property for the analysis. First the notion of well-formedness must be extended to take exception frames into account:

**Definition 3.21 (Extended Well-Formedness).** *A call-stack* $SF = F :: \langle m_2,pc_2,L_2,S_2\rangle :: \cdots :: \langle m_n,pc_n,L_n,S_n\rangle$ *is* extended well-formed *if and only if*

$$
\begin{array}{l}
\forall i \in \{2,\ldots,n\} : \\
\quad m_i.\mathit{instructionAt}(pc_i) = \texttt{invokestatic}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: S_i'' \wedge \\
\qquad m_{i-1} = m_i'' \\
\quad m_i.\mathit{instructionAt}(pc_i) = \texttt{invokevirtual}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: \mathit{loc}_i :: S_i'' \wedge \\
\qquad H(\mathit{loc}_i).\mathit{class} = \sigma_i \wedge \\
\qquad m_{i-1} = \mathit{methodLookup}(m_i'',\sigma_i) \\
\quad m_i.\mathit{instructionAt}(pc_i) = \texttt{invokeinterface}\ m_i'' \Rightarrow \\
\qquad S_i = v_{i,1} :: \cdots :: v_{i,|m_{i-1}|} :: \mathit{loc}_i :: S_i'' \wedge \\
\qquad H(\mathit{loc}_i).\mathit{class} = \sigma_i \wedge \\
\qquad m_{i-1} = \mathit{methodLookup}(m_i'',\sigma_i)
\end{array}
$$

*and either* $F = \langle m_1,pc_1,L_1,S_1\rangle$ *or* $F = \langle \mathsf{Exc}\ \mathit{loc}_X,(m_1,pc_1)\rangle$ *for some* $\mathit{loc}_X$.

Intuitively the definition states that a call stack is extended well-formed if either it is well-formed, see Definition 3.10, or if the top of the call stack is an exception frame and the rest of it is well-formed. Next Lemma 3.11 is extended to also hold for Carmel$_{\mathsf{EXC}}$:

**Lemma 3.22.** *For $P \in \mathsf{Program}$ if $\langle K, H, SF\rangle$ is an extended well-formed configuration and $P \vdash \langle K, H, SF\rangle \Longrightarrow_{EXC} \langle K', H', SF'\rangle$ then $\langle K', H', SF'\rangle$ is extended well-formed.*

*Proof.* Simple adaptation of the proof for Lemma 3.11 since exceptions only give rise to exception frames and cannot create entirely new "ordinary" stack frames. ∎

In addition to the representation function for exceptions, cf. equation (3.111), all the representation functions defined for the control flow analysis can be re-used without modification. Even the correctness relation for stack frames can be re-used since exception frames are handled at the level of call stacks:

$$\begin{array}{l} F_1 :: \cdots F_n \; \mathcal{R}^H_{\mathsf{EXC,CallStack}}\; (\hat{L}, \hat{S}, \hat{E}) \quad \text{iff} \\ \quad \forall i \in \{2, \ldots, n\} \colon F_i \; \mathcal{R}^H_{\mathsf{Frame}}\; (\hat{L}, \hat{S}) \wedge \\ \quad F_1 = \langle m_1, pc_1, L_1, S_1\rangle \;\Rightarrow\; F_1 \; \mathcal{R}^H_{\mathsf{Frame}}\; (\hat{L}, \hat{S}) \wedge \\ \quad F_1 = \langle \mathsf{Exc}\ loc_X, (m_X, pc_X)\rangle \;\Rightarrow\; \beta^H_{\mathsf{Ref}}(loc_X) \in \hat{E}(m_X) \end{array}$$

Note that only the top frame in a call stack can be an exception frame. The correctness relation for semantic configurations can then be defined as follows:

$$\begin{array}{lll} \langle K, H, SF\rangle \; \mathcal{R}_{\mathsf{EXC,Conf}}\; (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) & \text{iff} & \beta^H_{\mathsf{StaHeap}}(K) \sqsubseteq \hat{K} \wedge \\ & & \beta_{\mathsf{Heap}}(H) \sqsubseteq \hat{H} \wedge \\ & & SF \; \mathcal{R}^H_{\mathsf{EXC,CallStack}}\; (\hat{L}, \hat{S}, \hat{E}) \end{array}$$

Following the same procedure as for the control flow analysis a subject reduction property for the exception analysis can now be stated and proved:

**Theorem 3.23 (Subject Reduction (EXC)).** *Assume $P \in \mathsf{Program}$, such that $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{EXC} P$ and let $C$ be an extended well-formed semantic configuration such that $P \vdash C \Longrightarrow_{EXC} C'$ then*

$$C \; \mathcal{R}_{\mathsf{EXC,Conf}}\; (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \;\Rightarrow\; C' \; \mathcal{R}_{\mathsf{EXC,Conf}}\; (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E})$$

*Proof.* The proof is an extension of the proof for Theorem 3.14 (suject reduction for the control flow analysis).

*Case* `numop`*:* Assume that $C = \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle$ and that

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\mathrm{EXC}} (m, pc) : \texttt{numop}\ \ t\ \ op \tag{3.112}$$

There are three sub-cases.

*Sub-case 1:* If the instruction does not throw an exception the case proceeds exactly as in the proof for Theorem 3.14.

*Sub-case 2:* If the instruction throws an exception that is handled locally it must be the case that

$$C' = \langle K, H, \langle m, pc_0, L, loc_X :: \epsilon\rangle :: SF\rangle$$

for some $pc_0$ such that $\mathit{findHandler}(m, pc, \texttt{ArithmeticExc}) = pc_0$. It then follows from equation (3.112) that

$$\mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \texttt{ArithmeticExc}), (m, pc))$$

and thus by definition of the handler predicate that:

$$\{(\mathsf{Ref}\ \texttt{ArithmeticExc})\} :: \epsilon \sqsubseteq \hat{S}(m, pc_0)$$
$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$$

and thus $\beta^H_{\mathsf{LocHeap}}(L) \sqsubseteq \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$ whence

$$\langle m, pc_0, L, loc_X :: \epsilon\rangle\ \mathcal{R}^H_{\mathsf{Frame}}\ (\hat{L}, \hat{S})$$

This sub-case now follows.

*Sub-case 3:* If the instruction throws an exception that is not handled locally. Then it follows that

$$C' = \langle K, H, \langle \mathsf{Exc}\ loc_X, (m_X, pc_X)\rangle :: \langle m_2, pc_2, L_2, S_2\rangle :: SF\rangle$$

with $H(loc_X).class = \texttt{ArithmeticExc}$ and such that no local handler is found: $\mathit{findHandler}(m, pc, \texttt{ArithmeticExc}) = \bot$. Thus from equation (3.112) and the definition of the analysis and the exception predicate it must be the case that

$$(\mathsf{Ref}\ \texttt{ArithmeticExc}) \in \hat{E}(m)$$

From which this sub-case follows.

This concludes the case for `numop`.

*Case "local handler":* Assume that

$$C = \langle K, H, \langle \mathsf{Exc}\ loc_X, (m_X, pc_X)\rangle :: \langle m_2, pc_2, L_2, S_2\rangle :: SF\rangle$$

with $\sigma_X = H(loc_X).class$ such that

$$\mathit{findHandler}(m_2, pc_2, \sigma_X) = pc_X \tag{3.113}$$

and, by assumption, $\beta^H_{\mathsf{Ref}}(loc_X) \in \hat{E}(m_X)$; thus

$$C' = \langle K, H, \langle m_2, pc_X, L_2, loc_X :: \epsilon\rangle :: SF\rangle$$

From the extended well-formedness of $C$ it follows that the instruction at $(m_2, pc_2)$ is a method invocation. Assume, without loss of generality that $m_2.\mathit{instructionAt}(pc_2) = \texttt{invokevirtual}\ m_0$ then $S_2 = v_1 :: \cdots :: v_{|m_0|} :: loc :: S_2'$ and by assumption

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\mathrm{EXC}} (m_2, pc_2) : \texttt{invokevirtual}\ m_0 \tag{3.114}$$

with $m_X = \mathit{methodLookup}(m_0, H(\mathit{loc}).\mathit{class})$ since $C$ is extended well-formed. From equation (3.114) it follows that

$$\forall(\mathsf{Ref}\ \sigma_X) \in \hat{E}(m_X) : \mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X), (m_2, pc_2))$$

and thus from equation (3.113):

$$\{(\mathsf{Ref}\ \sigma_X)\} :: \epsilon \sqsubseteq \hat{S}(m_2, pc_X)$$
$$\hat{L}(m_2, pc_2) \sqsubseteq \hat{L}(m_2, pc_X)$$

and thus the sub-case follows.

The remaining cases are similar. ■

An interesting thing to note about the exception analysis is that it was built simply by extending the control flow analysis with an exception handling predicate rather than starting from scratch. The proof of correctness was also done by re-using the proof for the control flow analysis with added proof-cases specific to the exception semantics. This indicates a certain degree of robustness of the analysis.

### 3.5.4 Implementation

The exception analysis has been implemented in a prototype analysis and verification tool. The implementation was obtained by simply extending the implementation of the control flow analysis (Section 3.4) with predicates for the exception handler and an exception component. A simple, but verbose, encoding of the *findHandler* function was chosen: for each line of the program in which an exception may be raised it specifies the correct handler. This encoding must be generated before solving the constraints and thus requires a preprocessing step.

## 3.6 Summary

In this chapter a control flow analysis for the Carmel language was designed, specified, and formally proved correct with respect to the semantics. Furthermore, it was shown how the Flow Logic specification for the control flow analysis systematically can be turned into a constraint generator over ALFP which in combination with the Succinct Solver provides a way to efficiently implement the control flow analysis. The constraint generator was formally proved to be equivalent to the Flow Logic specification. Finally the control flow analysis was extended to track the set of exceptions that a program may throw during evaluation. The resulting exception analysis also formally proved correct with respect to the semantics.

The control flow analysis specified in this chapter has been formalised and formally proved correct using the Coq proof assistant as reported in [CJPR04].

Proofs in the Coq system are constructive and thus programs "implementing" the proofs may be extracted via the Curry-Howard isomorphism. Extracting the program corresponding to the proof of correctness of the analysis results in an implementation of the analysis that is correct by construction. It is noted in [CJPR04, Section 7] that the Coq formalisation was facilitated by the analysis being specified in the Flow Logic framework.

A rather different approach to analysing bytecode is taken in the SOOT framework for analysing and optimising Java bytecode, cf. [VRCG$^+$99]. There the Java bytecode is converted into a sequence of intermediate formats (called Baf, Jimple[VRH98], and Grimp respectively) and different analyses and optimisations are performed at each stage. Baf is described as a "streamlined representation" of Java bytecode, Jimple as a typed three-address representation, and Grimp as an "aggregated version" of Jimple, cf. [VRCG$^+$99, VRGH$^+$00]. An important feature of the Jimple and Grimp intermediate representations is that the operand stack has been removed and stack access converted to equivalent operations on local variables; this approach makes it possible to use standard analysis techniques, e.g., monotone frameworks, to be applied more readily. In contrast the flexibility of the Flow Logic framework allows for a natural formulation of analyses directly on the stack based bytecode.

Both [PCC01] and [SJ03b] define frameworks for abstract interpretation of Java and a simple imperative object-oriented language respectively. Both frameworks can be instantiated to perform various analyses of the target language.

In [TP00] a number of different propagation based call graph analyses (basically control flow analyses) are surveyed and compared on speed and precision. For efficiency reasons, most of the analyses model the operand stack and local variables by simply merging the operand stack (local variables) for all instructions.

Chapter 4

# Extending the Flow Logic

*I object to doing things that computers can do*
*—Olin Shivers*

The control flow and exception analyses defined in the previous chapter provide essential information on the basic behaviour of a program. In this chapter various additions and extensions are explored and how they can be accommodated by the fundamental analyses, with the twofold intention of increasing the scope of the analyses and their applications, and to demonstrate the malleability and robustness of the analyses.

## 4.1 Data Flow Analysis

In addition to computing information about the possible control flow of a program it is often desirable or even necessary to gain more detailed information about the data flow of a program. Knowledge of the data flow in a program is useful for a wide variety of applications, spanning from optimisation to debugging and verification, but can also be used to enhance the precision of the control flow analysis. In this section a relatively simple data flow analysis, based on *modification counts*, is defined and illustrate how the additional information gained can be used to improve the precision of both the control flow and the exception analyses defined in the previous chapter. In Section 5.1 the data flow analysis plays an integral part in the *transaction flow analysis* used for verifying the well-formedness of transactions.

Data in a Carmel program is comprised of numbers, object references, and return addresses. Since the latter two are already tracked by the control flow analysis the data flow analysis need only track numbers and the operations on them. The analysis defined here can be viewed as a variation over the

$$
\begin{array}{lcl}
\mathsf{Val}_{\mathsf{DFA}} & = & \mathsf{Num}_{\mathsf{DFA}} + \mathsf{Ref} + \mathsf{RetAddr} \\
\mathsf{Object}_{\mathsf{DFA}} & = & (\mathit{class} : \mathsf{Class}) \times \\
 & & (\mathit{fieldValue} : \mathsf{Field} \to \mathsf{Val}_{\mathsf{DFA}}) \\
\mathsf{Array}_{\mathsf{DFA}} & = & (\mathit{type} : \mathit{Type}) \times \\
 & & (\mathit{length} : \mathbb{N}_0) \times \\
 & & (\mathit{value} : \mathbb{N}_0 \to \mathsf{Val}_{\mathsf{DFA}}) \\
\mathsf{StaHeap}_{\mathsf{DFA}} & = & \mathsf{Field} \to \mathsf{Val}_{\mathsf{DFA}} \\
\mathsf{LocHeap}_{\mathsf{DFA}} & = & \mathbb{N}_0 \to \mathsf{Val}_{\mathsf{DFA}} \\
\mathsf{Stack}_{\mathsf{DFA}} & = & \mathsf{Val}_{\mathsf{DFA}}^*
\end{array}
$$

Figure 4.1: Updated concrete domains

well-known *constant propagation* analysis, cf. [NNH99]. The underlying idea is that the exact numbers of interest are tracked through a (small) number of modifications after which they are simply considered to be "unknown", i.e., mapped to the top-value of the analysis domain. Such an analysis is very well-suited for tracking constants in JCVML programs since they are often calculated (form other constants) in the beginning of a program and then left unmodified for the rest of the program.

The data flow analysis is defined for the entire $\text{Carmel}_{\mathsf{EXC}}$ language.

### 4.1.1 Instrumenting the Semantics

While most of the semantics defined in Chapter 2 can be re-used directly it is necessary to instrument the numerical values with *modification counts* in order to prove the correctness of the data flow analysis. The instrumentation merely records the modification counts of numbers and does not change the underlying semantics in any way.

First the domain for recording the instrumented numbers must be defined. Modification counts can only take on the values of the natural numbers and zero which leads naturally to the following domain for instrumented numbers:

$$\mathsf{Num}_{\mathsf{DFA}} = \mathbb{Z} \times \mathbb{N}_0$$

The idea being that an instrumented number, $(z, \chi) \in \mathsf{Num}_{\mathsf{DFA}}$, is a number, $z$, that has been modified at most $\chi$ times during program execution. To make the instrumented semantics more readable instrumented numbers are written as $z^{\chi}$.

While only the numbers domain needs to be changed in any substantial way to account for the instrumentation, all the domains that depend on numbers must be updated to reflect the changed numbers domains. The updated domains are shown in Figure 4.1.

With the domains in place the semantic rules must also be updated to reflect the new domains and in particular the new domain for numbers. The changes

are simple and straightforward and only the rules that involve numbers directly need to be changed. Below the necessary changes are illustrated for a few rules.

Constants are assumed to start with a modification count of 0. This is evident in the rule for `push`:

$$\frac{m.\mathit{instructionAt}(pc) = \texttt{push}\ \ t\ \ c \qquad v = \begin{cases} \texttt{null} & \text{if } c = \texttt{null} \\ c^0 & \text{otherwise} \end{cases}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

When numbers are used in computations the modification counts must be increased, as exemplified by the rule for (binary) numerical operations: in order to compute the modification count for the result the maximum is taken of the modification counts of the arguments and add one (for the current operation):

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{numop}\ \ t\ \ \mathit{op}\ \ [t'] \\ \mathit{op} \in \mathsf{BinaryOp} \qquad \mathit{op} \in \{\texttt{div}, \texttt{rem}\} \Rightarrow v_2 \neq 0 \\ \chi = \max\{\chi_1, \chi_2\} + 1 \qquad v = \mathit{applyBinary}(\mathit{op}, v_1, v_2) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v_1^{\chi_1} :: v_2^{\chi_2} :: S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \\ \langle K, H, \langle m, pc+1, L, v^{\chi} :: S\rangle :: SF\rangle \end{array}}$$

In Figure 4.2 the adapted rules are shown. The remaining rules need no modification to handle the instrumented numbers.

### 4.1.2 Abstract Domains

For the analysis an abstract representation of the instrumented numbers is chosen that allows tracking of numbers through finitely many modifications. Numbers that have been modified more than $N$ number of times are represented by the top value, written $\mathsf{INT}_N$:

$$\overline{\mathsf{Num}}^{\top}_N = (\mathbb{Z} \times \{0, \ldots, N\}) \cup \{\mathsf{INT}_N\}$$

Notice how this actually defines a *family* of abstract domains parameterised by the maximum number of modifications, $N$, tracked by the domain. This enables an easy fine-tuning of the precision of the data flow analysis on a case-by-case basis.

Because the control flow analysis modelled all numbers simply as a single value, there was no need to model arithmetic operators since they would always result in the same value. This is no longer the case due to the increased precision achieved by the modification counts. Therefore the analysis must also model the arithmetic operators. For $\mathit{op} \in \mathsf{BinOp}$ and $z_1^{\chi_1}, z_2^{\chi_2} \in \overline{\mathsf{Num}}^{\top}_N \setminus \{\mathsf{INT}_N\}$ the abstract arithmetic operator is defined as follows:

$$\delta_{op}(z_1^{\chi_1}, z_2^{\chi_2}) = \begin{cases} \mathit{op}(z_1, z_2)^{\max\{\chi_1,\chi_2\}+1} & \text{if } \max\{\chi_1, \chi_2\} < N \\ \mathsf{INT}_N & \text{otherwise} \end{cases}$$

and $\delta_{op}(z^{\chi}, \mathsf{INT}_N) = \delta_{op}(\mathsf{INT}_N, z^{\chi}) = \delta_{op}(\mathsf{INT}_N, \mathsf{INT}_N) = \mathsf{INT}_N$. The abstract arithmetic function for unary operators is defined in a similar manner. The

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ t\ \ c}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L, c^0 :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{inc}\ \ t\ \ x\ \ c \\ v_1^{\chi_1} = L(x) \qquad v = v_1 + c \qquad \chi = \chi_1 + 1 \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L[x \mapsto v^{\chi}], S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{BinaryOp} \qquad op \in \{\texttt{div}, \texttt{rem}\} \Rightarrow v_2 \neq 0 \\ \chi = \max\{\chi_1, \chi_2\} + 1 \qquad v = applyBinary(op, v_1, v_2) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v_1^{\chi_1} :: v_2^{\chi_2} :: S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \\ \langle K, H, \langle m, pc+1, L, v^{\chi} :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{UnaryOp} \qquad \chi = \chi_1 + 1 \qquad v = applyUnary(op, v_1) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v_1^{\chi_1} :: S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L, v^{\chi} :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{instanceof}\ \ \sigma \\ v = \begin{cases} 1 & \text{if } H(loc).class \preceq \sigma \\ 0 & otherwise \end{cases} \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L, v^0 :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arraylength} \qquad loc \neq \texttt{null} \qquad v = H(loc).length}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{DFA}} \langle K, H, \langle m, pc+1, L, v^0 :: S\rangle :: SF\rangle}$$

Figure 4.2: Instrumented semantics

$$
\begin{array}{lcl}
\overline{\mathsf{Val}}_{\mathsf{DFA}} & = & \overline{\mathsf{Num}}_N^{\top} + \overline{\mathsf{Ref}} + \overline{\mathsf{RetAddr}} \\
\widehat{\mathsf{Val}}_{\mathsf{DFA}} & = & \mathcal{P}(\overline{\mathsf{Val}}_{\mathsf{DFA}}) \\
\widehat{\mathsf{Object}} & = & \mathsf{Field} \to \widehat{\mathsf{Val}}_{\mathsf{DFA}} \\
\widehat{\mathsf{Array}} & = & \widehat{\mathsf{Val}}_{\mathsf{DFA}} \\
\widehat{\mathsf{LocHeap}}_{\mathsf{DFA}} & = & \mathsf{Addr} \to \mathbb{N}_0 \to \widehat{\mathsf{Val}}_{\mathsf{DFA}} \\
\widehat{\mathsf{Stack}}_{\mathsf{DFA}} & = & \overline{\mathsf{Addr}} \to (\widehat{\mathsf{Val}}_{\mathsf{DFA}}^{*})^{\top} \\
\widehat{\mathsf{StaHeap}} & = & \mathsf{Field} \to \widehat{\mathsf{Val}}_{\mathsf{DFA}} \\
\widehat{\mathsf{Analysis}}_{\mathsf{DFA}} & = & \widehat{\mathsf{StaHeap}}_{\mathsf{DFA}} \times \widehat{\mathsf{Heap}}_{\mathsf{DFA}} \times \widehat{\mathsf{LocHeap}}_{\mathsf{DFA}} \times \widehat{\mathsf{Stack}}_{\mathsf{DFA}} \times \widehat{\mathsf{ExcCache}}
\end{array}
$$

Figure 4.3: Updated abstract domains

above is easily extended to operate on sets of abstract values $A_1, A_2 \in \widehat{\mathsf{Val}}$:

$$\delta_{op}(A_1, A_2) = \left\{ \delta_{op}(a_1, a_2) \,\middle|\, a_1 \in A_1 \cap \overline{\mathsf{Num}}_N^{\top}, a_2 \in A_2 \cap \overline{\mathsf{Num}}_N^{\top} \right\}$$

The abstract arithmetic function for unary operators is extended in a similar fashion. For $op \in \mathsf{UnaryOp}$ and $z_1^{\chi_1} \in \overline{\mathsf{Num}}_N^{\top} \setminus \{\mathsf{INT}_N\}$ define

$$\delta_{op}(z_1^{\chi_1}) = \begin{cases} op(z_1)^{\chi_1+1} & \text{if } \chi_1 < N \\ \mathsf{INT}_N & \text{otherwise} \end{cases}$$

which is extended to operate on sets, let $A_1 \in \widehat{\mathsf{Val}}$:

$$\delta_{op}(A_1) = \left\{ \delta_{op}(a_1) \,\middle|\, a_1 \in A_1 \cap \overline{\mathsf{Num}}_N^{\top} \right\}$$

As was the case for the semantic domains, changing the basic domain for (abstract) numbers implies that all domains depending on the numbers domain must be updated to reflect that change. Figure 4.3 shows the updated abstract domains.

### 4.1.3 Flow Logic Specification

With the abstract domains and operators in place the data flow analysis can now be specified as a small extension of the control flow analysis. Since the control flow analysis is specified in a way that is entirely independent on the specific data flow of a program, it only depends on the fact that there is a flow, it is sufficient to adapt the clauses for instructions that actually work with numbers: `push`, `numop`, `inc`, `instanceof`, `new` (default values), `arraylength`.

Before proceeding with the specification the representation function for constants is adapted to take the updated domains into account:

$$\beta_{\mathsf{DFA},\mathsf{Const}}(c) = \begin{cases} \beta_{\mathsf{Num}_{\mathsf{DFA}}}(c, 0) & \text{if } c \in \mathsf{Num} \\ \{\mathtt{null}\} & \text{if } c = \mathtt{null} \\ \beta_{\mathsf{RetAdr}}(c) & \text{if } c \in \mathsf{RetAdr} \end{cases}$$

Obviously only the case for numbers need to be changed: it now uses the proper representation function (as defined in the next section) and uses a default modification count of zero since constants that occur directly in a program, by definition, have not been modified yet.

Applying the above representation function the new specification for the instructions `push` and `instanceof` becomes:

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{push}\ t\ c\\ \quad \text{iff} \quad \beta_{\mathsf{DFA,Const}}(c) :: \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{instanceof}\ t\\ \quad \text{iff} \quad A :: X \triangleleft \hat{S}(m,pc) :\\ \qquad (\beta_{\mathsf{DFA,Const}}(0) \cup \beta_{\mathsf{DFA,Const}}(1)) :: X \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

Using the abstract arithmetic functions the analysis of numerical operations is updated in the following way:

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{numop}\ t\ \mathit{unop}\ [t']\\ \quad \text{iff} \quad A :: X \triangleleft \hat{S}(m,pc) :\\ \qquad \delta_{\mathit{unop}}(A) :: X \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{numop}\ t\ \mathit{binop}\ [t']\\ \quad \text{iff} \quad A_1 :: A_2 :: X \triangleleft \hat{S}(m,pc) :\\ \qquad \delta_{\mathit{binop}}(A_1,A_2) :: X \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

For the `new` instruction the only change needed is to re-define the *default* function, that maps fields to their (abstract) default value, to use the new representation function:

$$\forall f \in \mathit{instanceFields}(\sigma):\ \ \mathit{default}_{\text{DFA}}(\sigma)(f) = \beta_{\mathsf{DFA,Const}}(\mathit{def}(f.\mathit{type}))$$

The specification is updated accordingly:

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{new}\ \sigma\\ \quad \text{iff} \quad \{(\mathsf{Ref}\ \sigma)\} :: \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \mathit{default}_{\text{DFA}}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

Since array lengths are not tracked, the `arraylength` instruction just returns the top value of the numerical domain:

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{arraylength}\\ \quad \text{iff} \quad B :: X \triangleleft \hat{S}(m,pc) :\\ \qquad \{\mathsf{INT}_N\} :: \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc+1)\\ \qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1)\end{array}$$

Finally, for the `inc`-instruction the abstract arithmetic operator is used to calculate the new value of the local variable $x$ by adding the constant $n$:

$$\delta_{add}(\hat{L}(m,pc)(x), \beta_{\mathsf{DFA,Const}}(n))$$

which then gives rise to the following updated specification:

$$\begin{array}{l}(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{inc}\ \ t\ \ x\ \ n \\ \quad \text{iff} \quad \hat{S}(m,pc) \sqsubseteq \hat{S}(m,pc+1) \\ \qquad\quad \delta_{add}(\hat{L}(m,pc)(x), \beta_{\mathsf{DFA,Const}}(n)) \sqsubseteq \hat{L}(m,pc+1)(x) \\ \qquad\quad \hat{L}(m,pc) \sqsubseteq_{\{x\}} \hat{L}(m,pc+1)\end{array}$$

The Flow Logic specification for the rest of the instructions remains unchanged. Note that the data flow analysis as specified here merely tracks numerical values through a number of modifications. In Section 4.1.5 the results of the data flow analysis are used to increase the precision of the control flow analysis and the exception analysis.

### 4.1.4 Semantic Correctness

Proving the semantic correctness of the data flow analysis is quite easy and follows the same technique as for the previous analyses. First a representation function for the new numbers domain is introduced. In the following is assumed that a particular $N$ has been chosen for the abstract numbers domain $\overline{\mathsf{Num}}^{\top}_{N}$:

$$\beta_{\mathsf{Num_{DFA}}}(z^{\chi}) = \begin{cases} \{z^{\chi}\} & \text{if } n \leq N \\ \{\mathsf{INT}_N\} & \text{otherwise} \end{cases}$$

Next the representation functions and correctness relations that depend on the representation function for numbers are updated. First the representation function for values:

$$\beta^{H}_{\mathsf{DFA,Val}}(v) = \begin{cases} \beta_{\mathsf{Num_{DFA}}}(v) & \text{if } v \in \mathsf{Num_{DFA}} \\ \beta^{H}_{\mathsf{Ref}}(v) & \text{if } v \in \mathsf{Ref} \\ \beta_{\mathsf{RetAdr}}(v) & \text{if } v \in \mathsf{RetAdr} \end{cases}$$

where $H \in \mathsf{Heap}$ and $v \in \mathsf{Val}$. Since none of the other representation functions need to be changed, except to use the new representation function for values, the details are elided here. The correctness relations are also unchanged from those used for the control flow analysis. Thus the semantic correctness of the data flow analysis can now be stated and proved. First observe that the extended well-formedness lemma trivially extends to cover the instrumented semantics:

**Lemma 4.1.** *If $P \in \mathsf{Program}$ and $\langle K, H, SF\rangle$ is an extended well-formed configuration and $P \vdash \langle K, H, SF\rangle \Longrightarrow_{DFA} \langle K', H', SF'\rangle$ then $\langle K', H', SF'\rangle$ is extended well-formed.*

*Proof.* Trivial since the semantics for method invocation and return is unchanged. ∎

Now the subject reduction property can be established, and thus semantic correctness, for the data flow analysis:

**Theorem 4.2 (Subject Reduction (DFA)).** *Assume that* $P \in \mathsf{Program}$*, such that* $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{DFA} P$ *and let* $C = \langle K, H, SF\rangle$ *be an extended well-formed semantic configuration such that* $P \vdash C \Longrightarrow_{DFA} C'$ *then*

$$C\ \mathcal{R}_{\mathsf{DFA,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \Rightarrow C'\ \mathcal{R}_{\mathsf{DFA,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E})$$

*Proof.* Analogous to the proof of Theorem 3.14; only the modified rules need to be proved. Below only the case for `numop` is shown in detail.

*Case* `numop`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{BinaryOp} \qquad op \in \{\texttt{div}, \texttt{rem}\} \Rightarrow v_2 \neq 0 \\ \chi = \max\{\chi_1, \chi_2\} + 1 \qquad v = applyBinary(op, v_1, v_2) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v_1^{\chi_1} :: v_2^{\chi_2} :: S\rangle :: SF\rangle \Longrightarrow_{\mathrm{DFA}} \\ \langle K, H, \langle m, pc+1, L, v^{\chi} :: S\rangle :: SF\rangle \end{array}}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \models_{\mathrm{DFA}} (m, pc) : \texttt{numop}\ \ t\ \ op\ \ [t'] \tag{4.1}$$

and

$$\langle K, H, \langle m, pc, L, v_1^{\chi_1} :: v_2^{\chi_2} :: S\rangle :: SF\rangle\ \mathcal{R}_{\mathsf{DFA,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}) \tag{4.2}$$

It follows from equation (4.1) that

$$\beta^H_{\mathsf{DFA}}(K) \sqsubseteq \hat{K} \tag{4.3}$$

$$\beta_{\mathsf{DFA}}(H) \sqsubseteq \hat{H} \tag{4.4}$$

$$\beta^H_{\mathsf{DFA}}(L) \sqsubseteq \hat{L}(m, pc) \tag{4.5}$$

$$\beta^H_{\mathsf{DFA}}(v_1^{\chi_1} :: v_2 ::^{\chi_2} :: S) \sqsubseteq \hat{S}(mp, c) \tag{4.6}$$

From the definition of the analysis and equations (4.1) and (4.5) it follows that

$$\begin{array}{rcl} \beta^H_{\mathsf{DFA}}(L) & \sqsubseteq & \hat{L}(m, pc) \\ & \sqsubseteq & \hat{L}(m, pc+1) \end{array} \tag{4.7}$$

Also from the definition of the analysis and equation (4.1) it is the case that $A_1 :: A_2 :: X \triangleleft \hat{S}(m, pc)$ and thus $\beta^H_{\mathsf{DFA}}(v_1^{\chi_1}) \sqsubseteq A_1$ and $\beta^H_{\mathsf{DFA}}(v_2^{\chi_2}) \sqsubseteq A_2$ and therefore by definition that

$$\delta_{op}(v_1^{\chi_1}, v_2^{\chi_2}) :: X \sqsubseteq \hat{S}(m, pc+1) \tag{4.8}$$

If $\max\{\chi_1, \chi_2\} \geq N$ the case follows trivially; assume therefore that $\max\{\chi_1, \chi_2\} < N$ then, by definition:

$$\begin{array}{rcl} v^{\chi} & = & op(v_1, v_2)^{\max\{\chi_1, \chi_2\}+1} \\ & \in & \delta_{op}(v_1^{\chi_1}, v_2^{\chi_2}) \end{array}$$

The case now follows from equations (4.3), (4.4), (4.7), and (4.8).

The remaining cases are similar or trivial. ∎

While some work was needed to instrument the semantics and adapt all the semantic and abstract domains and their representation functions this was straightforward and mostly trivial. This was also the case for the analysis specification and proof of correctness where only modified clauses actually had to be specified and proved. This is a very important aspect of the analysis, made possible through the use of the Flow Logic framework, that allows for an analysis to be developed incrementally.

### 4.1.5 Using the Data Flow Analysis

In Section 5.1 the data flow analysis defined in this chapter is used as part of an analysis that ensures the well-formedness of transactions. In this section different ways of using the data flow analysis to improve the precision of the previously discussed control flow analysis and exception analysis are considered.

#### Improving the Control Flow Analysis

One of the traditional uses of a data flow analysis is to enhance the precision of the underlying control flow analysis. In the following the control flow analysis of Section 3.2 is adapted to make use of the data flow component to improve the precision of the control flow analysis. This is done by exploiting the knowledge gained from the data flow analysis about the possible data values that can flow into a branching instruction, e.g., an `if`-instruction, since branches that are never executed need not be analysed. To make this work an abstract evaluation of the branching condition must be performed to determine when the condition *may* evaluate to true. This is analogous to modelling the abstract arithmetic operators. Since only numbers and references can be meaningfully compared in Carmel two predicates for abstract evaluation of conditions are defined: one for numbers and one for references.

For (abstract) numbers the possibility of a top-value, $\mathsf{INT}_N$, must be taken into account. If one of the operands is or contains $\mathsf{INT}_N$ then the conditional may succeed immediately. Otherwise the conditional may succeed only if there exists operands for which the comparison operator evaluates to true:

$$\begin{array}{l}\mathsf{COND}_{\mathsf{Num}}(cmp, A_1, A_2) \equiv \\ \quad (\mathsf{INT}_N \in A_1) \vee (\mathsf{INT}_N \in A_2) \vee \\ \quad (\exists z_1^{\chi_1} \in A_1 : \exists z_2^{\chi_2} \in A_2 : cmp(z1, z2))\end{array}$$

where $cmp \in \{\mathtt{eq}, \mathtt{ne}, \mathtt{gt}, \mathtt{ge}, \mathtt{lt}, \mathtt{le}\}$. By substituting $\neg cmp$ for $cmp$ this predicate can also be used to evaluate when a comparison may fail:

$$\begin{array}{l}\mathsf{COND}_{\mathsf{Num}}(\neg cmp, A_1, A_2) \equiv \\ \quad (\mathsf{INT}_N \in A_1) \vee (\mathsf{INT}_N \in A_2) \vee \\ \quad (\exists z_1^{\chi_1} \in A_1 : \exists z_2^{\chi_2} \in A_2 : \neg cmp(z1, z2))\end{array}$$

For references the situation is even simpler because references can only be compared for equality and non-equality. The lack of a top-value for abstract object

references simplifies matters further. Since concrete object references are represented by their respective classes in the analysis, it follows that if two concrete references are equal (in the sense of the `eq` comparison operator) then their corresponding abstract references will be identical. However, the converse, determining if two object references are unequal, is not possible from the abstract object references alone since there might be two instances of the same object leading to two different locations with the same abstract representation. This leads to the following predicate for abstract evaluation of reference comparison:

$$\begin{aligned}&\mathsf{COND}_{\mathsf{Ref}}(cmp, A_1, A_2) \equiv\\ &\quad (cmp = \texttt{eq} \Rightarrow A_1 \cap A_2 \neq \emptyset) \wedge\\ &\quad (cmp = \texttt{ne} \Rightarrow true)\end{aligned}$$

Note that comparison of references is only defined for $cmp \in \{\texttt{eq}, \texttt{ne}\}$. The converse predicate is then defined as follows:

$$\begin{aligned}&\mathsf{COND}_{\mathsf{Ref}}(\neg cmp, A_1, A_2) \equiv\\ &\quad (cmp = \texttt{eq} \Rightarrow true) \wedge\\ &\quad (cmp = \texttt{ne} \Rightarrow A_1 \cap A_2 \neq \emptyset)\end{aligned}$$

To simplify the analysis specification the predicates defined above are combined into one, parameterised on the operand type, cf. Section 2.2.10, of the conditional:

$$\begin{aligned}&\mathsf{COND}_t(cmp, A_1, A_2) \equiv\\ &\quad (t \in \{\texttt{s}, \texttt{b}, \texttt{i}\} \Rightarrow \mathsf{COND}_{\mathsf{Num}}(cmp, A_1, A_2)) \wedge\\ &\quad (t \in \{\texttt{r}\} \Rightarrow \mathsf{COND}_{\mathsf{Ref}}(cmp, A_1 \cap \overline{\mathsf{ObjRef}}, A_2 \cap \overline{\mathsf{ObjRef}}))\end{aligned}$$

This predicate also ensures that when it is used to compare references then only the subsets of $A_1$ and $A_2$ containing object references are compared. The predicate for comparing numbers implicitly ensures that only numbers are used for comparison by only quantifying over constants of the form $(z, n)$. All that remains to be done is to add the combined predicate to the analysis specification for conditionals in such a way that a branch is not analysed if it cannot possibly be reached in the semantics. The updated Flow Logic specification for conditionals is shown in Figure 4.4. The analysis specifications of `lookupswitch` and `tableswitch` are amended in a similar manner.

**Remark 4.3.** *A close inspection of the proof of Theorem 4.2 (subject reduction for the data flow analysis) for the amended instructions is actually sufficient for also proving the correctness of the amended data flow analysis since the abstract condition predicate is a conservative approximation of the concrete conditions.*

### Improving the Exception Analysis

Building on the amended data flow analysis of the previous section the precision of the exception analysis defined in Section 3.5 can also be improved. Here improved precision means reducing the number of spurious or false exceptions that are reported due to the approximative nature of the analysis.

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0 \\
\quad \text{iff} \quad A_1 :: A_2 :: X \lhd \hat{S}(m,pc) : \\
\qquad \mathsf{COND}_t(\neg cmp, A_1, A_2)\ \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\qquad \mathsf{COND}_t(cmp, A_1, A_2)\ \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m,pc_0) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc_0)
\end{array}
$$

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{if}\ \ t\ \ cmp\ \ nul\ \ \texttt{goto}\ \ pc_0 \\
\quad \text{iff} \quad A :: X \lhd \hat{S}(m,pc) : \\
\qquad \mathsf{COND}_t(\neg cmp, A, \{nul\})\ \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\qquad \mathsf{COND}_t(cmp, A, \{nul\})\ \Rightarrow \\
\qquad\quad X \sqsubseteq \hat{S}(m,pc_0) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc_0)
\end{array}
$$

Figure 4.4: Updated Flow Logic specification for conditionals

The numerical operators `div` (division) and `rem` (remainder) provide a good example since they throw an `ArithmeticExc` exception if division by zero is attempted. Since the exception analysis must provide a sound approximation to the actual exceptions thrown it is assumed that numerical operations always may throw an `ArithmeticExc` exception. However, using the data flow analysis combined with the abstract conditionals from the previous section it is possible to check whether division by zero is actually possible and only then throw an (abstract) exception. This is formalised in the following Flow Logic clause:

$$
\begin{array}{l}
(\hat{K},\hat{H},\hat{L},\hat{S},\hat{E}) \models_{\text{DFA}} (m,pc) : \texttt{numop}\ \ t\ \ binop\ \ [t'] \\
\quad \text{iff} \quad A_1 :: A_2 :: X \lhd \hat{S}(m,pc) : \\
\qquad \delta_{binop}(A_1,A_2) :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\qquad binop \in \{\texttt{div},\texttt{rem}\} \wedge \mathsf{COND}_{\mathsf{Num}}(\texttt{eq},\{0^0\},A_2)\ \Rightarrow \\
\qquad\quad \mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \texttt{ArithmeticExc}),(m,pc))
\end{array}
$$

where the abstract condition predicate, $\mathsf{COND}_{\mathsf{Num}}$, is used to determine if the divisor can actually be zero. Note that this approach can trivially be adapted to also check against `null`-references in instructions that work on arrays or objects.

The formal statement and proof of correctness is omitted. Similar to the amended data flow analysis defined in the previous section a close inspection of the proof for the ordinary data flow analysis suffices, cf. Remark 4.3.

### 4.1.6 Implementation

Even though the Succinct Solver lacks convenient, built-in support for arithmetic operations it is still possible and feasible to implement the data flow analysis for small maximal modification counts. Such an implementation requires a "manual" modelling of all the pertinent arithmetic operators and quickly becomes unwieldy. The transaction flow analysis discussed in Section 5.1 builds on a data flow analysis with a maximal modification count of one and has been implemented in a prototype analysis and verification tool.

A better alternative would be to use a constraint solver that has better support for arithmetic, e.g., XSB Prolog. As part of the performance benchmark of XSB Prolog and Succinct Solver described in [Pil03] an automatic conversion from ALFP constraints to equivalent XSB constraints was developed. It would therefore be quite simple to use XSB for solving ALFP(-like) constraint possibly extended with arithmetic.

## 4.2 Applet Firewall and Ownership Analysis

The Java Card platform is a so-called *multi-applet* platform which means that several different applets may run on the same Java smart card, i.e., on the same Java Card virtual machine. However, with this added flexibility also comes an increased risk in terms of safety and security because now applets may (try to) interfere with each other or even attack each other. To prevent this from happening the Java Card platform isolates each applet in a designated area and controls access to other areas through the notion of *object ownership* and an *applet firewall* mechanism, cf. [Che00, SJE01]. While total isolation of individual applets is ideal seen from a security perspective it defeats one of the purposes, and most interesting uses, of a multi-applet platform, namely to let different applets cooperate on certain tasks. To (re-)enable cooperation it is possible to selectively "punch holes" in the firewall and thereby allow objects of specified classes to be *shared* between applets.

In this section the semantics of Carmel is extended, based on the approach in [Han02a], to incorporate object ownership and the applet firewall. It does not cover the full runtime system and details concerning the special runtime protocols used when setting up object sharing, cf. Section 4.2.1, are modelled in a more abstract way. By similarly enhancing the basic control flow analysis with basic ownership information and *ownership analysis* is obtained.

### 4.2.1 Ownership and Sharing

In JCVML and Carmel an applet is *owned* by the package in which it is defined. Additionally certain special objects and arrays are owned by the runtime system, usually denoted JCRE short for "Java Card Runtime Environment", making them available to all other owners. Each applet is executed in a specific *ownership context* corresponding to the owner of the applet. The ownership

context of the applet currently executing is called the *current ownership context*. At any point in time there is only one such current ownership context. Any objects and arrays created by an applet similarly belong to the ownership context of the creating applet. To protect applets from harming each other the *applet firewall* prevents applets from one ownership context to access or (directly) communicate with applets in a different ownership context. To allow applets from different contexts to cooperate several ways exist in JCVML for applets to share objects and information, mainly through the use of system owned arrays or objects. These mechanisms are best suited for exchanging data between applets or between an applet and the runtime system. Since these mechanisms rely on special low level JCRE-specific mechanisms they are not included in the present model. A detailed formalisation is given in [SJE01].

For more complex interaction between applets the concept of *sharing* provides a flexible alternative that allows one applet to indicate to the firewall that some of the methods of a certain class may be invoked by objects belonging to another context. Specifically a class must implement a special interface, the `Shareable` interface, or a sub-interface thereof to indicate that some or all of the methods in objects of that class may be invoked by "outsiders". Only the methods mentioned in the sub-interface are tagged as accessible from other contexts. These special methods are then accessed through the `invokeinterface` instruction and not through ordinary virtual method invocation. In order for the runtime system to perform its tasks an exception is made: methods executing on behalf of the runtime system, in the JCRE context, are exempted from scrutiny by the firewall and can thus access both fields and methods of both shared and ordinary objects in any other context. In order to invoke a method in a shared object a reference to that object is needed. However since communication between applets, other than through shared objects, is prohibited by the firewall a special mechanism bypassing the firewall is needed for an applet to obtain such a reference. In JCVML this is handled through special methods and objects owned and implemented by the runtime system exploiting the special access bestowed on methods executing in the JCRE context. In order to avoid modelling the details of the runtime environment a simpler scheme is implemented for sharing: entry point methods, cf. Section 2.4.2, are invoked with references to shared objects (in other contexts) as parameters. Thus, sharing is initially set up when program execution starts. See Section 4.2.3 for details.

The applet firewall acts as a simple filter on which methods can be invoked across ownership boundaries. No such filtering is done on the applet that is invoking a method across a boundary: *any* applet that has a reference to a shared object may invoke the (shared) methods in that object. In essence the firewall only protects objects that are not shared. One way to guard against unwanted access to a shared object is for an applet to try and ensure that only the intended applets get a reference to the shared object. In JCVML this is accomplished by using a much simplified version of the *stack inspection* technique known from Java and the Java Virtual Machine. For Java Card and JCVML stack inspection amounts to checking that the last owner (different from the current) is allowed access which is sufficient for most purposes. However,

once the object reference is obtained there is nothing to prevent an applet from accidentally or intentionally leaking the reference to another applet not intended by the original applet. To overcome this problem it is customary, at least for security aware applets, to perform the above check not only before handing out an object reference but *every time* the object reference is used to invoke a shared method.

In Section 6 an analysis and a *devil's advocate* are designed that can statically guarantee that certain object references are never leaked even in the presence of malicious applets; thereby obviating the need to know the whole program, including possible attackers, beforehand.

### 4.2.2 Adding Ownership to the Semantic Domains

In JCVML there are two kinds of ownership contexts: individual packages and the special "JCRE" context indicating that an object is owned by the runtime system and thus accessible to everyone. This leads to the following simple domain for ownership information:

$$\mathsf{Owner} = \mathsf{PackageName} + \{\mathsf{JCRE}\}$$

This then leads to the new domain for objects that includes ownership information:

$$\begin{aligned}\mathsf{Object}_{\mathsf{OWN}} = {} & (\mathit{class} : \mathsf{Class}) \times \\ & (\mathit{owner} : \mathsf{Owner}) \times \\ & (\mathit{fieldValue} : \mathsf{Field} \to \mathsf{Val})\end{aligned}$$

and similarly for arrays:

$$\begin{aligned}\mathsf{Array}_{\mathsf{OWN}} = {} & (\mathit{type} : \mathsf{ElemType}) \times \\ & (\mathit{length} : \mathbb{N}_0) \times \\ & (\mathit{owner} : \mathsf{Owner}) \times \\ & (\mathit{value} : \mathbb{N}_0 \to \mathsf{Val})\end{aligned}$$

Finally each method invocation occurs on behalf of an owner. Thus an owner is added to every stack frame:

$$\mathsf{Frame}_{\mathsf{OWN}} = \mathsf{Owner} \times \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocHeap} \times \mathsf{Stack}$$

These are the only domains that are changed in a non-trivial way. Domains that directly or indirectly depend on these are trivially updated to reflect the new domain as shown in Figure 4.5.

### 4.2.3 Semantic Rules

Below the reduction rules are discussed for the semantics with ownership information added. Only the rules that require non-trivial modifications are listed since the remaining instructions only reads and updates local information.

$$
\begin{array}{lcl}
\mathsf{Heap}_{\mathsf{OWN}} & = & \mathsf{Ref} \to (\mathsf{Object}_{\mathsf{OWN}} + \mathsf{Array}_{\mathsf{OWN}}) \\
\mathsf{CallStack}_{\mathsf{OWN}} & = & (\mathsf{Frame}_{\mathsf{OWN}} + \mathsf{ExcFrame}) \times \mathsf{Frame}^{*}_{\mathsf{OWN}} \\
\mathsf{RunConf}_{\mathsf{OWN}} & = & \mathsf{StaHeap} \times \mathsf{Heap}_{\mathsf{OWN}} \times \mathsf{CallStack}_{\mathsf{OWN}} \\
\mathsf{FinConf}_{\mathsf{OWN}} & = & \mathsf{StaHeap} \times \mathsf{Heap}_{\mathsf{OWN}} \times \mathsf{Val}_{\perp}
\end{array}
$$

Figure 4.5: Semantic domains with ownership

Therefore the semantics of these instruction only requires a simple addition to track the current owner context. In general rules of the form:

$$P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K', H, \langle m, pc', L', S'\rangle :: SF\rangle$$

are replaced with rules of the form:

$$P \vdash \langle K, H, \langle own, m, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\mathrm{OWN}} \langle K', H, \langle own, m, pc', L', S'\rangle :: SF\rangle$$

Since the instructions only access local or static information neither the heap nor the call stack is changed.

For instructions that may access global information, e.g., through method invocation or instance field manipulation, the virtual machine inserts runtime checks to ensure that the requested access is in fact allowed. As discussed above methods that are executed on behalf of the runtime system, i.e., owned by JCRE, can access everything without restriction whereas other methods can only access objects that they themselves own. See Section 4.2.1 for more details. The access control policy is expressed in the semantics as the following predicate:

$$\mathit{checkOwner}(own, o) \equiv (own = \mathsf{JCRE}) \ \vee \ (own = o.\mathit{owner}) \tag{4.9}$$

In JCVML a failed ownership check would result in a `SecurityExc` exception being thrown. Here the semantics is simply stuck when an ownership test fails. This is consistent with the treatment of other runtime exceptions that are also not covered in Carmel. However, security exceptions can be added rather easily since the mechanisms for user defined exceptions are already present to handle the `throw` instruction.

### Initial Configurations

As already mentioned, sharing in JCVML is set up by a trivial, if tedious, protocol involving the runtime environment. While there are no real technical difficulties in modelling this aspect of the runtime environment it would not add any particular insights to the problems of sharing and ownership. Instead the initial sharing (necessary to cross the firewall the first time) is integrated into the initial configurations and require that, in addition to a self reference, entry point methods are invoked with references to the shared objects they are allowed to access as parameters.

Since only main classes are instantiated in an initial configuration all object references must point to an object of a main class; furthermore since each applet can only be instantiated once (in the initial configuration) is suffices to define sharing based on (main) classes. Note that this set up does not prohibit other forms of "manual" sharing later in the program execution; it only affects the initial sharing needed to cross the firewall the first time.

The function *sharing* : $\mathsf{Program} \times \mathsf{Class} \rightarrow \mathsf{Class}^*$ is used to specify the initial sharing: for each (main) class, $\sigma$, it returns a list of (main) classes that $\sigma$ are allowed to communicate with; this list is called the *initial knowledge* of $\sigma$.

Initial configurations can now be formally defined in the following way:

**Definition 4.4 (Initial configurations).** *For a program,* $P \in \mathsf{Program}$*, the configuration* $C \in \mathsf{RunConf}$ *is an* initial configuration *if and only if* $\sigma \in P.main$*,* $m_\sigma = \sigma.entry$*,* $\sigma.sharing = \sigma_1 :: \cdots :: \sigma_n$*,* $\sigma_i \in P.main$ *for* $1 \leq i \leq n$*,* $C = \langle K, H, \langle m_\sigma, 0, [0 \mapsto loc_\sigma, 1 \mapsto loc_{\sigma_1}, \ldots, n \mapsto loc_{\sigma_n}], \epsilon\rangle :: \epsilon\rangle$ *and* $\forall \tau \in P.main : \exists loc_\tau : H(loc_\tau) = \tau$*.*

In Chapter 6 the initial knowledge of a class is used to define the idea of *leaked references* and an analysis is designed to prevent references from being leaked.

### Object Fragment

The updated semantics of the object fragment is shown in Figure 4.6. The modifications are discussed in some detail below.

A new object is owned by its creator, i.e., by the current ownership context. The *newObject* function must be updated to take this into account:

$$\begin{aligned} &newObject : \mathsf{Class} \times \mathsf{Owner} \times \mathsf{Heap} \rightarrow \mathsf{Location} \times \mathsf{Heap} \\ &newObject(\sigma, own, H) = (loc, H') \end{aligned}$$

where

$$\begin{aligned} &loc \notin \mathrm{dom}(H) \wedge o \in \mathsf{Object} \wedge H' = H[loc \mapsto o] \wedge \\ &o.class = \sigma \wedge H(loc).owner = own \end{aligned}$$

As before, the instance fields of a newly created object must be initiated with the correct default values:

$$\forall f \in \sigma.fields : \neg f.isStatic \Rightarrow o.fieldValue(f) = def(f.type)$$

The runtime type checks, `checkcast` and `instanceof`, supported by Carmel need to take into account whether a target class can be accessed from the current context either directly or through sharing. This is formalised as the following predicate:

$$\begin{aligned} checkShareable(own, o, iface) \equiv\ & checkOwner(own, o) \vee \\ & (iface \in \mathsf{Interface} \wedge isShareable(o.class, iface)) \end{aligned}$$

where the *isShareable* determines if a given class implements the `Shareable` interface and whether or not access happens through (a sub-interface of) the

$$\frac{m.instructionAt(pc) = \texttt{new}\ \sigma \qquad (loc, H') = newObject(\sigma, own, H)}{P \vdash \langle K, H, \langle own, m, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H', \langle own, m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{checkcast}\ \sigma \quad loc \neq \texttt{null} \Rightarrow (H(loc).class \preceq \sigma \wedge checkShareable(own, H(loc), \sigma))}{P \vdash \langle K, H, \langle own, m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle own, m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{instanceof}\ \sigma \quad v = \begin{cases} 1 & \text{if } loc \neq \texttt{null} \wedge H(loc).class \preceq \sigma \\ 0 & otherwise \end{cases} \quad loc \neq \texttt{null} \Rightarrow checkShareable(own, H(loc), \sigma)}{P \vdash \langle K, H, \langle own, m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle own, m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{getfield}\ f \quad loc \neq \texttt{null} \qquad o = H(loc) \qquad v = o.fieldValue(f) \qquad checkOwner(own, o)}{P \vdash \langle K, H, \langle own, m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle own, m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{putfield}\ f \qquad loc \neq \texttt{null} \qquad o = H(loc) \quad o' = o[fieldValue \mapsto o.fieldValue[f \mapsto v]] \qquad checkOwner(own, o)}{P \vdash \langle K, H, \langle own, m, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H[loc \mapsto o'], \langle own, m, pc+1, L, S\rangle :: SF\rangle}$$

Figure 4.6: Ownership semantics: object fragment

`Shareable` interface:

$$isShareable(\sigma, iface) \equiv \texttt{Shareable} \in \sigma.implements \wedge \texttt{Shareable} \in iface.super^*$$

Accessing an instance field, whether for reading or writing, is only allowed if the field is located in an object that is owned by the current context or if the current context is JCRE. The above predicate, *checkOwner*, formalising ownership check used as access control mechanism is re-used for the `getfield` and `putfield` instructions.

While ownership contexts control all access to instance fields no such control is extended to static fields or methods. This entails that only the already mentioned trivial additions are needed for the semantics of `getstatic` and `putstatic`.

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokestatic}\ m_0 \\ n = |m_0| \qquad L_0 = v_1 :: \cdots :: v_n \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle own, m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \\ \quad \langle K, H, \langle own, m_0, 0, L_0, \epsilon\rangle :: \langle own, m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ m_0 \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ L_v = loc :: v_1 \cdots :: v_{|m_0|} \qquad checkOwner(own, o) \\ m_v = methodLookup(m_0, o.class) \qquad F = \langle o.owner, m_v, 0, L_v, \epsilon\rangle \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \\ \quad \langle K, H, F :: \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokeinterface}\ m_0 \qquad loc \neq \texttt{null} \\ o = H(loc) \qquad L_v = loc :: v_1 \cdots :: v_{|m_0|} \qquad checkShareable(own, o, m_0.class) \\ m_v = methodLookup(m_0, o.class) \qquad F = \langle o.owner, m_v, 0, L_v, \epsilon\rangle \end{array}}{\begin{array}{l} P \vdash \langle K, H, \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \\ \quad \langle K, H, F :: \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \end{array}}$$

Figure 4.7: Ownership semantics: method fragment

### Method Fragment

The updated semantics for the method fragment is displayed in Figure 4.7.

Static methods are executed in the same context as the invoking method and therefore require no modifications. It is included in Figure 4.7 for completeness.

In contrast virtual methods are executed in the context of the object owning the method. Thus invoking a virtual method in an object that belongs to a different owner is not allowed. This is formalised using the *CheckOwner* predicate defined in equation (4.9) on page 107. However, if a class implements the `Shareable` interface the methods that are members of that class may be invoked by other owners, cf. 4.2.1.

As discussed in Section 4.2.1 interfaces provide the only way to invoke virtual methods in an object that belongs to another context. And then only if the invoked interface is a sub-interface of `Shareable` and the class of the target object implements the `Shareable` interface. This was formalised above as the *checkShareable* predicate also used for `invokeinterface`.

### Arrays

The ownership semantics for instructions in the arrays fragment is shown in Figure 4.8. As for objects newly created arrays are owned by their creator and the *newArray* function is updated to reflect this:

$$\begin{array}{l} newArray : \mathsf{Type} \times \mathbb{N} \times \mathsf{Owner} \times \mathsf{Heap} \to \mathsf{Location} \times \mathsf{Heap} \\ newArray(t, n, own, H) = (loc, H') \end{array}$$

$$\frac{m.instructionAt(pc) = \texttt{new (array } t\texttt{)} \quad n \geq 0 \quad (H', loc) = newArray(t, n, own, H)}{P \vdash \langle K, H, \langle own, m, pc, L, n :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H', \langle own, m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arraylength} \quad loc \neq \texttt{null} \quad n = H(loc).length \quad checkOwner(own, a)}{P \vdash \langle K, H, \langle own, m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle own, m, pc+1, L, n :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arrayload } t \quad loc \neq \texttt{null} \quad v = H(loc).value(n) \quad 0 \leq n < H(loc).length \quad checkOwner(own, a)}{P \vdash \langle K, H, \langle own, m, pc, L, n :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle own, m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{arraystore } t \quad loc \neq \texttt{null} \quad a = H(loc) \quad a' = a[value \mapsto a.value[n \mapsto v]] \quad H' = H[loc \mapsto a'] \quad 0 \leq n < H(loc).length \quad checkOwner(own, a)}{P \vdash \langle K, H, \langle own, m, pc, L, v :: n :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H', \langle own, k, pc+1, L, S\rangle :: SF\rangle}$$

Figure 4.8: Ownership semantics: arrays

where

$$loc \notin \text{dom}(H) \wedge a \in \mathsf{Array} \wedge H' = H[loc \mapsto a] \wedge$$
$$a.type = t \wedge a.length = n \wedge H(loc).owner = own$$

Furthermore, the array is initialised according to its type: $\forall i \in \{0, \ldots, n-1\} : a.value(i) = def(t)$.

The remaining array instructions are all controlled by the ownership check implemented by the *checkOwner* function of equation (4.9).

#### Exceptions

Only objects that are accessible to the current method may be thrown as an exception and thus an ownership check is implemented, again using the *checkOwner* predicate. Once the exception is thrown the notion of ownership is ignored when trying to find a handler. The updated semantics for the `throw` instruction is displayed in Figure 4.9.

### 4.2.4 Abstract Domains for Ownership Analysis

The ownership analysis described in this and the following sections approximates for each abstract object the set of possible owners and for each method the set

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{throw} \\ loc_X \neq \texttt{null} \qquad H(loc_X).class \preceq \texttt{Throwable} \qquad checkOwner(own, H(loc_X)) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc_X :: S\rangle :: SF\rangle \Longrightarrow_{\text{OWN}} \langle K, H, \langle \mathsf{Exc}\ loc_X\rangle :: SF\rangle}$$

Figure 4.9: Ownership semantics: exceptions

of possible contexts in which it is executed. It is designed as an extension of the basic control flow analysis defined in Section 3.2 and follows the extension of the semantics quite closely.

Abstract owners are modelled simply as a set of concrete owners:

$$\widehat{\mathsf{Owner}} = \mathcal{P}(\mathsf{Owner})$$

ordered by subset inclusion

$$\widehat{own}_1 \sqsubseteq \widehat{own}_2 \quad \text{iff} \quad \widehat{own}_1 \subseteq \widehat{own}_2$$

A further component is needed in the Flow Logic specification that tracks ownership information for every method, i.e., which contexts the given method may be invoked and executed in:

$$\widehat{\mathsf{OwnerCache}} = \mathsf{Method} \to \widehat{\mathsf{Owner}}$$

Finally an ownership component is added to the abstract objects reflecting a similar change in the concrete domain:

$$\begin{array}{rl} \widehat{\mathsf{Object}}_{\mathsf{OWN}} = & (\mathit{fieldValue} : \mathsf{Field} \to \widehat{\mathsf{Val}}) \times \\ & (\mathit{owner} : \widehat{\mathsf{Owner}}) \end{array}$$

which leads to the following ordering on abstract objects

$$\begin{array}{rll} \hat{o}_1 \sqsubseteq \hat{o}_2 & \text{iff} & \mathrm{dom}(\hat{o}_1.\mathit{fieldValue}) \subseteq \mathrm{dom}(\hat{o}_2.\mathit{fieldValue}) \wedge \\ & & \forall f \in \mathrm{dom}(\hat{o}_1.\mathit{fieldValue}) : \hat{o}_1.\mathit{fieldValue}(f) \subseteq \hat{o}_2.\mathit{fieldValue}(f) \wedge \\ & & \hat{o}_1.\mathit{owner} \sqsubseteq \hat{o}_2.\mathit{owner} \end{array}$$

Having changed the definition of abstract objects the definition of abstract heaps also needs to be updated:

$$\widehat{\mathsf{Heap}}_{\mathsf{OWN}} = (\overline{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}_{\mathsf{OWN}}) \times (\overline{\mathsf{ArrRef}} \to \widehat{\mathsf{Array}})$$

Using the same notational conventions as defined in Section 3.2.2.

The preceding modifications then lead to the following definition of the abstract domain for ownership analyses:

$$\widehat{\mathsf{Analysis}}_{\mathsf{OWN}} = \widehat{\mathsf{StaHeap}} \times \widehat{\mathsf{Heap}}_{\mathsf{OWN}} \times \widehat{\mathsf{LocHeap}} \times \widehat{\mathsf{Stack}} \times \widehat{\mathsf{ExcCache}} \times \widehat{\mathsf{OwnerCache}}$$

This domain is sufficient to specify an ownership analysis that over approximates the set of owners an object may have and the contexts in which a method may be executed.

### 4.2.5 Flow Logic Specification

The Flow Logic judgements for ownership analysis are of the general form:

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{instr}$$

where $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \in \widehat{\mathsf{Analysis}}_{\mathsf{OWN}}$.

The ownership analysis leverages the ownership controls to increase precision of the underlying control flow analysis in the same way the data flow analysis was used in Section 4.1.5. To this end an abstract ownership check is defined:

$$\widehat{\mathit{checkOwner}}(O_1, O_2) \equiv (\mathsf{JCRE} \in O_1) \vee (O_1 \cap O_2 \neq \emptyset)$$

This predicate models the *checkOwner*, cf. equation (4.9) on page 107, from the semantics. The approximative nature of the analysis implies that even in situations where the ownership controls would prohibit access to a given object the analysis may yet analyse the program as if access was allowed.

In Section 4.2.7 the ownership analysis is used to verify that no security violations can occur.

#### Object Fragment

The Flow Logic specification for the ownership analysis of the instruction in the object fragment is shown in Figure 4.10 and briefly discussed below.

The analysis for the `new` instruction must now make sure to update the set of possible owners with the possible owners of the current method.

For the runtime type-checks a formalisation of the *checkShareable* predicate is needed:

$$\begin{aligned}\widehat{\mathit{checkShareable}}(O_1, O_2, \sigma, \mathit{iface}) \equiv\ & \widehat{\mathit{checkOwner}}(O_1, O_2) \vee \\ & (\mathit{iface} \in \mathsf{Interface} \wedge \mathit{isShareable}(\sigma, \mathit{iface}))\end{aligned}$$

Using the above predicate the Flow Logic clauses for runtime type checks, implemented by the `checkcast` and `instanceof` instructions, can then be defined.

The analysis of static field access remains unchanged while instance field access must be controlled by the $\widehat{\mathit{checkOwner}}$ predicate.

#### Method Fragment

Figure 4.11 shows the specification for the ownership analysis of the method fragment.

The clause for invoking static methods, `invokestatic`, need only a simple modification, the invoked method should inherit the context of the invoker. Strictly speaking it is not necessary for the analysis to track ownership contexts for static methods. However even if the runtime firewall does not define ownership for static fields and methods the information computed by the analysis may be used to implement as statically checked access control for static fields and methods based on ownership.

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{new}\ \sigma$
iff $\{(\mathsf{Ref}\ \sigma)\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\mathit{default}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma).\mathit{fieldValue}$
$\hat{O}(m) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma).\mathit{owner}$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{checkcast}\ \tau$
iff $B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B\colon \widehat{\mathit{checkShareable}}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).\mathit{owner}, \sigma, \tau) \Rightarrow$
$\{(\mathsf{Ref}\ \sigma)\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{instanceof}\ \tau$
iff $B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B\colon \widehat{\mathit{checkShareable}}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).\mathit{owner}, \sigma, \tau) \Rightarrow$
$\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{getfield}\ f$
iff $B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B\colon \widehat{\mathit{checkOwner}}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).\mathit{owner}) \Rightarrow$
$(\hat{H}(\mathsf{Ref}\ \sigma)(f)) :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{putfield}\ f$
iff $A :: B :: X \triangleleft \hat{S} :$
$\forall(\mathsf{Ref}\ \sigma) \in B\colon \widehat{\mathit{checkOwner}}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).\mathit{owner}) \Rightarrow$
$A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma).\mathit{fieldValue}(f)$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

Figure 4.10: Ownership analysis: object fragment

As for instance field access, virtual method invocation is controlled by the $\widehat{checkOwner}$ predicate and in addition the ownership cache for the invoked method must include the current possible contexts.

Interfaces are handled very much like virtual methods except that additional checks are made to see if access is allowed by the rules for sharing. This is done using the $\widehat{checkShareable}$ predicate defined above.

#### Arrays

The analysis for arrays is very similar to that for objects and instance fields. The specification for analysis of arrays is given in Figure 4.12.

As for objects, whenever a new array is created the set of possible owners must be updated to contain also the set of current possible contexts. The instructions for accessing arrays are simply enhanced with abstract context checks.

#### Exceptions

For user thrown objects a context check, using $\widehat{checkOwner}$, is added to ensure that only accessible objects are used for exceptions. The specification is given in Figure 4.13.

### 4.2.6 Semantic Correctness

Analogous to the previous analyses, semantic correctness of the ownership analysis can be proved by establishing a subject reduction property based on a number of representation functions. For clarity and brevity the definition of domains and functions that have not been changed or only changed in a trivial way are omitted.

Only one extra representation function is needed: the one for owners. Since the domain of abstract owners is defined to be a set of concrete owners the representation function is defined simply as the injection of a concrete owner into a set:

$$\beta_{\mathsf{Owner}}(own) = \{own\}$$

Next the representation function for objects must be changed to reflect the changes in the underlying domain:

$$\beta^H_{\mathsf{OWN,Object}}(o) = [\mathit{fieldValue} \mapsto \beta^H_{\mathsf{Val}} \circ o.\mathit{fieldValue}, \mathit{owner} \mapsto \beta_{\mathsf{Owner}}(o.\mathit{owner})]$$

Finally the correctness relation should take ownership information, i.e., the current context, into account:

$$\langle own, m, pc, L, S\rangle \; \mathcal{R}^H_{\mathsf{OWN,Frame}} \; (\hat{L}, \hat{S}, \hat{O}) \quad \text{iff} \quad \begin{array}{l} \beta^H_{\mathsf{LocHeap}}(L) \sqsubseteq \hat{L}(m, pc) \wedge \\ \beta^H_{\mathsf{Stack}}(S) \sqsubseteq \hat{S}(m, pc) \wedge \\ \beta_{\mathsf{Owner}}(own) \sqsubseteq \hat{O}(m) \end{array}$$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{invokestatic}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: X \triangleleft \hat{S}(m, pc)$ :
$A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_0, 0)[0..|m_0| - 1]$
$\forall \sigma_X \in \hat{E}(m_0) : \mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X), (m, pc))$
$\hat{O}(m) \sqsubseteq \hat{O}(m_0)$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{invokevirtual}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc)$ :
$\forall(\mathsf{Ref}\ \sigma) \in B : \widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner) \Rightarrow$
$m_v \triangleleft methodLookup(m_0, \sigma)$ :
$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$
$\forall \sigma_X \in \hat{E}(m_v)$: $\mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X), (m, pc))$
$\hat{H}(\mathsf{Ref}\ \sigma).owner \sqsubseteq \hat{O}(m_v)$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{invokeinterface}\ m_0$
iff $A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc)$ :
$\forall(\mathsf{Ref}\ \sigma) \in B$ :
$\widehat{checkShareable}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner, \sigma, m_0.class) \Rightarrow$
$m_v \triangleleft methodLookup(m_0, \sigma)$ :
$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$
$\forall \sigma_X \in \hat{E}(m_v)$: $\mathsf{HANDLE}_{(\hat{L},\hat{S},\hat{E})}((\mathsf{Ref}\ \sigma_X), (m, pc))$
$\hat{H}(\mathsf{Ref}\ \sigma).owner \sqsubseteq \hat{O}(m_v)$
$m_0.returnType = \texttt{void} \Rightarrow$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$m_0.returnType \neq \texttt{void} \Rightarrow$
$A :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : A :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$

Figure 4.11: Ownership analysis: method fragment

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{new (array } t\texttt{)}$
iff $A :: X \triangleleft \hat{S}(m, pc) :$
$\{(\mathsf{Ref}\ (\texttt{array}\ t))\} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$
$\hat{O}(m) \sqsubseteq \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{arraylength}$
iff $B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B :$
$\widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \Rightarrow$
$\{\mathsf{INT}\} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{arrayload } t$
iff $A :: B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B:$
$\widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \Rightarrow$
$\hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)) :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{arraystore } t$
iff $A_1 :: A_2 :: B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B:$
$\widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \Rightarrow$
$A_1 \sqsubseteq \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t))$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$

Figure 4.12: Ownership analysis: arrays

$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\text{OWN}} (m, pc) : \texttt{throw}$
iff $B :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma_X) \in B:$
$\widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma_X).owner) \Rightarrow$
$\mathsf{HANDLE}_{(\hat{L}, \hat{S}, \hat{E})}((\mathsf{Ref}\ \sigma_X), (m, pc))$

Figure 4.13: Ownership analysis: exceptions

Note that only regular stack frames, i.e., non-exception stack frames, need to be re-defined since ownership is ignored when finding a possible handler for an exception frame.

Before stating the subject reduction property for the ownership analysis it is observed that the extended well-formedness lemma also holds for the ownership semantics:

**Lemma 4.5.** *For $P \in \mathsf{Program}$ if $\langle K, H, SF\rangle$ is an extended well-formed configuration and $P \vdash \langle K, H, SF\rangle \Longrightarrow_{OWN} \langle K', H', SF'\rangle$ then $\langle K', H', SF'\rangle$ is extended well-formed.*

*Proof.* Extension of the proof for Lemma 3.22 since the semantics underlying method invocation has not changed. ∎

The proof is a trivial extension of the proof for Lemma 3.22 since the instrumented semantics does not provide new ways of creating stack frames.

Using the well-formedness lemma the following theorem can be proved stating the subject reduction property for the ownership analysis:

**Theorem 4.6 (Subject Reduction (OWN)).** *Let $P \in \mathsf{Program}$ be a program such that $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{OWN} P$ and let $C = \langle K, H, SF\rangle$ be an extended well-formed semantic configuration such that $P \vdash C \Longrightarrow_{OWN} C'$ then*

$$C\ \mathcal{R}_{\mathsf{OWN,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \Rightarrow C'\ \mathcal{R}_{\mathsf{OWN,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O})$$

*Proof.* Analogous to the proof of Theorem 3.14. The case for `invokeinterface` is shown in detail below.

*Case* `invokeinterface`*:* By assumption

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokeinterface}\ m_0 \qquad loc \neq \texttt{null} \\ o = H(loc) \qquad L_v = loc :: v_1 \cdots :: v_{|m_0|} \qquad F = \langle o.\mathrm{owner}, m_v, 0, L_v, \epsilon\rangle \\ checkShareable(own, o, m_0.class) \qquad m_v = methodLookup(m_0, o.class) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\mathrm{OWN}} \\ \langle K, H, F :: \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \end{array}}$$

and

$$(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{\mathrm{OWN}} (m, pc) : \texttt{invokevirtual}\ m_0 \tag{4.10}$$

and for $F_0 = \langle own, m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle$:

$$\langle K, H, F_0 :: SF\rangle\ \mathcal{R}_{\mathsf{OWN,Conf}}\ (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \tag{4.11}$$

From equation (4.11) it follows directly that

$$\beta^H(K) \sqsubseteq \hat{K} \tag{4.12}$$
$$\beta(H) \sqsubseteq \hat{H} \tag{4.13}$$
$$\beta^H(L) \sqsubseteq \hat{L}(m, pc) \tag{4.14}$$
$$\beta^H(v_1 :: \cdots :: v_{|m_0|} :: loc :: S) \sqsubseteq \hat{S}(m, pc) \tag{4.15}$$
$$\beta(own) \sqsubseteq \hat{O}(m) \tag{4.16}$$

By definition of the analysis, equation (4.10) implies that $\hat{S}(m, pc) = A_1 :: \cdots :: A_{|m_0|} :: B :: X$ for some $A_i$, $B$, and $X$. Thus from equation (4.16):

$$\begin{aligned} \beta^H(v_i) &\sqsubseteq A_i \qquad \text{for } i \in \{1, \ldots, |m_0|\} && (4.17)\\ \beta^H(loc) &\sqsubseteq B && (4.18)\\ \beta^H(S) &\sqsubseteq X && (4.19) \end{aligned}$$

Let $\sigma = o.class$ then $\beta^H(loc) = (\mathsf{Ref}\ \sigma)$ and therefore (by equation (4.13)):

$$\begin{aligned} \beta^H(o) &= \beta^H(H(loc))\\ &\sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma) \end{aligned}$$

and thus in particular:

$$\beta(o.owner) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma).owner \tag{4.20}$$

From the semantics it follows that the *checkShareable* predicate holds:

$$checkShareable(own, o, m_0.class)$$

which is equivalent to

$$\begin{aligned} &checkOwner(own, o)\ \vee\\ &(m_0.class \in \mathsf{Interface} \wedge isShareable(o.class, m_0.class)) \end{aligned}$$

Now assume that

$$checkOwner(own, o)$$

then by definition of *checkOwner*:

$$(own = \mathsf{JCRE})\ \vee\ (own = o.owner)$$

In either case equations (4.16) and (4.20) implies that the following predicate is true:

$$\widehat{checkOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner) \tag{4.21}$$

Now assume instead that the following is true:

$$(m_0.class \in \mathsf{Interface} \wedge isShareable(o.class, m_0.class))$$

Then, since $\sigma = o.class$, it is the case that

$$(m_0.class \in \mathsf{Interface} \wedge isShareable(\sigma, m_0.class)) \tag{4.22}$$

and therefore equations (4.18), (4.21), and (4.22) combined ensures that

$$\widehat{checkShareable}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner, \sigma, m_0.class) \tag{4.23}$$

Using (4.23) equations (4.10) and (4.18) gives that

$$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$$

where $m_v = methodLookup(m_0, \sigma)$; thus

$$\begin{aligned} \beta^H(L_v) &\sqsubseteq \hat{L}(m_v, 0) && (4.24)\\ \beta^H(\epsilon) &\sqsubseteq \hat{S}(m_v, 0) && (4.25)\\ \hat{H}(\mathsf{Ref}\ \sigma).owner &\sqsubseteq \hat{O}(m_v) && (4.26) \end{aligned}$$

Then from (4.20) and (4.26) it follows that

$$\begin{aligned} \beta(o.owner &\sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma).owner \\ &\sqsubseteq \hat{O}(m_v) \end{aligned} \qquad (4.27)$$

The case now follows from the equations (4.12), (4.13), (4.24), (4.25), and (4.27).

The remaining cases are similar or simpler. ■

### 4.2.7 Containment

In order for the analysis defined in the preceding sections to be semantically correct it must over approximate control flow and ownership information. In particular this may lead to situations where an abstract ownership check succeeds but the actual semantic ownership check fails. In this section it is briefly discussed how the analysis information readily can be used to guarantee that no semantic ownership checks fail. This property is called *containment*:

**Definition 4.7 (Containment).** *A program $P \in \mathsf{Program}$ is said to be* contained *if the execution of $P$ is never stuck because of a failed owner check.*

This is the equivalent of requiring that the program in question never throws a `SecurityExc` exception.

Next a set of conditions is defined that must hold of the analysis in order to guarantee containment. These conditions amount to nothing more than a (much) stricter version of the abstract ownership conditions already verified in the analysis. First a strict version of the $\widehat{checkOwner}$ predicate used in the analysis:

$$\begin{aligned} \widehat{strictOwner}(O_1, O_2) \equiv\ & O_1 = \{\mathsf{JCRE}\} \vee \\ & (|O_1| = |O_2| = 1) \wedge (O_1 = O_2) \end{aligned}$$

Note that $|O_1| = |O_2| = 1$ is required, due to the approximative nature of the analysis, to be able to meaningfully make the necessary comparison: $O_1 = O_2$. For the strict version it is not sufficient to check if $O_1 \cap O_2 \neq \emptyset$ since this does not guarantee that the two owners are identical. Instructions that may obtain access through sharing need an additional check:

$$\begin{aligned} \widehat{strictShareable}(O_1, O_2, \sigma, iface) \equiv\ & \widehat{strictOwner}(O_1, O_2) \vee \\ & (iface \in \mathsf{Interface} \wedge isShareable(\sigma, iface)) \end{aligned}$$

$$
\begin{array}{l}
\widehat{isContained}(P, (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O})) \equiv \\
\quad \forall \sigma \in P.classes : \forall m \in \sigma.methods : \\
\qquad m.instructionAt(pc) = \texttt{getfield}\ f \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner) \\
\qquad m.instructionAt(pc) = \texttt{putfield}\ f \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner) \\
\qquad m.instructionAt(pc) = \texttt{invokevirtual}\ m_0 \Rightarrow \\
\qquad\quad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner) \\
\qquad m.instructionAt(pc) = \texttt{checkcast}\ \tau \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictShareable}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner, \sigma, \tau) \\
\qquad m.instructionAt(pc) = \texttt{instanceof}\ \tau \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictShareable}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner, \sigma, \tau) \\
\qquad m.instructionAt(pc) = \texttt{invokeinterface}\ m_0 \Rightarrow \\
\qquad\quad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictShareable}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner, \sigma, m_0.class) \\
\qquad m.instructionAt(pc) = \texttt{arraylength} \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \\
\qquad m.instructionAt(pc) = \texttt{arrayload}\ t \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \\
\qquad m.instructionAt(pc) = \texttt{arraystore}\ t \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ (\texttt{array}\ t)) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ (\texttt{array}\ t)).owner) \\
\qquad m.instructionAt(pc) = \texttt{throw} \Rightarrow \\
\qquad\quad B :: X \triangleleft \hat{S}(m, pc) \wedge \\
\qquad\quad \forall(\mathsf{Ref}\ \sigma) \in B : \widehat{strictOwner}(\hat{O}(m), \hat{H}(\mathsf{Ref}\ \sigma).owner)
\end{array}
$$

Figure 4.14: Definition of the $\widehat{isContained}$ predicate

The formal specification of the verification conditions is shown in Figure 4.14 as a predicate that verifies certain aspects of an analysis.

It can now be stated and proved that the extended checking on the analysis is sufficient to guarantee containment.

**Proposition 4.8.** *Let* $P \in$ Program *such that* $(\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}) \models_{OWN} P$. *Then* $\widehat{isContained}(P, (\hat{K}, \hat{H}, \hat{L}, \hat{S}, \hat{E}, \hat{O}))$ *implies that* $P$ *is* contained.

*Proof.* Follows immediately from the soundness of the ownership analysis, cf. Theorem 4.6. ■

This result demonstrates some of the potential of the ownership analysis even if the additional checks are rather strict. In particular, if the $\widehat{isContained}$ predicate does not hold it can be used to pinpoint the location of a possible firewall violation and thus aid in debugging a program under construction or assist in understanding the behaviour of a program of unknown or untrusted origin.

### 4.2.8 Implementation

A prototype implementation of the ownership analysis and the containment property is briefly described in [Han02a]. The implementation is a straightforward extension of the prototype implementation of the control flow analysis discussed in Section 3.4.

## 4.3 Summary

In this chapter two extensions of the basic control flow and exception analyses of Chapter 3 were discussed in detail: a data flow analysis and an ownership analysis.

The data flow analysis (Section 4.1) demonstrates how traditional program analyses may be specified in a straightforward and modular way by re-using the foundation laid by the control flow and exception analyses. In Section 5.1 the data flow analysis (including the improved analysis of conditionals) is used to ensure that transactions are used in a manner that does not result in errors. Data flow analysis is an essential component of modern compilers where it is primarily used for optimisation purposes, cf. [ASU85, App98]. However, the Java (Card) bytecode verifier presents a prominent example of an alternative use of data flow analysis, namely program verification and validation, cf. [Sun00, LY99]. See [NNH99] for a textbook discussion of data flow analysis.

The ownership analysis (Section 4.2), also specified as a straightforward extension of the control flow and exception analyses, does not represent a traditional program analysis. Indeed, it is an analysis of a unique aspect of the JCVML semantics and the Java Card runtime environment. Furthermore, it illustrates how advanced and specialised features of JCVML can be added to the basic Carmel language in a natural and modular way that to a large extent allows both semantics and analyses to be re-used.

The notion of ownership and the applet firewall are discussed in [Che00] and formalised in [Siv04, SJE01]. Other approaches to verifying that a program does not violate the firewall policy include [MPH01, CHS01, ÉJ02]. In [MPH01] a type-system is defined for *applet isolation* that can statically ensure an applet will not try to access fields or methods of other applets. The type-system is developed for Java Card rather than JCVML. A constraint based solution to the same problem, but for (a small subset of) JCVML, can be found in [CHS01]. Here the constraints are used to describe the data flow in JCVML programs. Based on the solution to the constraints, instructions are classified with respect to their effect on the applet firewall ans possible violations are detected; this step is very similar to type checking. Another constraint based approach is discussed in [ÉJ02] where a control flow analysis is defined for JCVML (converted to a convenient three-address format similar to Jimple [VRH98, VRCG+99]) and formulated using *quantified conditional constraints*. Special instructions are added to the semantics to model the stack inspection mechanism and the low level protocols for exchanging references to shared objects.

Chapter 5

# Safety and Security

*Massive application of faith is not generally considered a valid security strategy*
*—David Bailey in [Bai95]*

Because of their unique physical properties smart cards are often used in safety and security critical applications that require a very high degree of assurance. For such applications it is crucial that special care is taken to protect the safety and security of programs and data on the card. However, the unpredictable way and environment in which smart cards are used combined with the somewhat arcane protocols and conventions used in Java Card programming makes it extraordinarily difficult to guarantee the safety and security of a program and its data.

In this chapter two analyses are presented: a Transaction Flow Analysis for verifying that transactions are used in a well-defined manner, and an Information Flow Analysis to ensure that secret information is not leaked. These analyses serve several purposes: first they are useful tools in their own right for ensuring that an applet has certain non-trivial desirable properties and thereby help alleviate the problems and reduce the workload inherent in verifying that an applet is indeed "well-behaved". Secondly, the development of these analyses demonstrate, in a larger perspective, the importance and intrinsic applicability of static analysis for validation and verification purposes.

## 5.1 Transaction Flow Analysis

Developing robust and fault-tolerant applications is a non-trivial task under optimal conditions. Smart cards rarely provide conditions that approach the optimal. The innate unpredictability of the (physical) runtime environment, e.g.,

a card may be torn from the card terminal during an update of critical data on the card, complicates matters further. To protect the integrity of critical data under such circumstances special programming techniques and methodologies must be employed. The JCVML API provides access for developers to an essential tool for maintaining data integrity and mitigating the effects of sudden disruption of data updates: *transactions*. Data updates taking place in a transaction are guaranteed atomicity, i.e., either all of the updates are successful or none of them are. This permits an application to recover gracefully from an acute interruption of program execution and thereby sustain a consistent state of critical data.

Processing and memory resources on a smart card are severely limited. The limited resources are apparent in the implementation of the JCVML transaction mechanism in that only one transaction can be active at any point in time and only a limited amount of data can be held in the commit buffer; violations result in an exception being thrown at runtime, i.e., if any attempt at opening a transaction is made while another transaction is active. Conversely an attempt at closing, either by committing or aborting, a transaction when no transaction is active also results in an exception. If no transaction on any possible execution path in a given program violates the above rules the transactions of the program are said to be *well-formed*. The limitation imposed by the above transaction semantics leads to a rather defensive programming style where it is explicitly checked if a transaction is in progress before starting a new one and similarly checking that a transaction is active before trying to close it. Furthermore, programmers often have to "work around" the limitation of not being able to nest transactions and this leads to a non-trivial control flow for transactions where a transaction may be started in one method and committed (or aborted) in another, possibly dependent upon whether or not the first method was invoked in a context where a transaction was already in progress or not.

The non-trivial flow of transactions and the defensive programming style conspire to make it very hard for a programmer to ensure that transactions are indeed well-formed since all possible program executions have to be taken into account. In this section the control flow analysis for Carmel Core is extended to a context dependent analysis using transaction status as context. It is then shown that the analysis can be used to prove that the transactions in a given program are well-behaved.

### 5.1.1 Extended Semantics

In this section the basic Carmel Core semantics is extended to model most of the transaction semantics of JCVML. The extended semantics given here is focused in modelling well-formedness aspects of transactions and thus no attempt is made to model the size aspects of the commit buffer. This is, in part, because the details concerning the size and the implementation of API calls relating to the size of the commit buffer are left unspecified and implementation dependent.

The JCVML transaction API is comprised by six methods. The first three are concerned with opening and closing a transaction: `beginTransaction`,

`commitTransaction`, `abortTransaction`. The fourth, `getTransactionDepth`, returns the number of currently open transactions, i.e., either 0 or 1. The last two, `getMaxCommitCapacity` and `getUnusedCommitCapacity`, are mentioned here only for completeness as they return the maximum size of the commit buffer and how much is still of available of the commit buffer respectively. For convenience the API calls are incorporated into Carmel Core directly as instructions:

$$\begin{array}{lll} \mathsf{Instruction} & ::= & \ldots \\ & \mid & \texttt{API.getTransactionDepth} \\ & \mid & \texttt{API.beginTransaction} \\ & \mid & \texttt{API.commitTransaction} \\ & \mid & \texttt{API.abortTransaction} \end{array}$$

Since the well-formedness property for transactions only depends on the order in which transactions are opened and closed it is, somewhat surprisingly, independent of the actual underlying transaction mechanism. This in turn implies that in order to establish the well-formedness property of a program, there is no need to track or even model actual transactions. Instead it is enough to keep track of the *transaction depth*, i.e., the number of currently active transactions which in the case of JCVML and Carmel Core is either 0 or 1.

The concrete semantic domains for Carmel Core only needs minor modifications to track the current transaction depth. The domain for transaction depths is quite trivial since only two states are allowed:

$$\mathsf{TransDepth} = \{0, 1\}$$

Transaction depths are denoted $\theta$. In anticipation of later uses and to facilitate the correctness proofs methods are annotated with the transaction depth in which they were invoked:

$$\mathsf{Method}_{\mathsf{TFA}} = \mathsf{Method} \times \mathsf{TransDepth}$$

An annotated method, $(m, \theta) \in \mathsf{Method}_{\mathsf{TFA}}$, is written $m^{\theta}$. Note that these annotations are different from the modification counts used for the data flow analysis, cf. Section 4.1, and thus $\theta$ is used instead of $\chi$. The current transaction depth is recorded as part of the stack frames:

$$\mathsf{Frame}_{\mathsf{TFA}} = \mathsf{TransDepth} \times \mathsf{Method}_{\mathsf{TFA}} \times \mathsf{PC} \times \mathsf{LocHeap} \times \mathsf{Stack}$$

The domains for call stacks and semantic configurations are merely updated to reflect the modified domains for stack frames:

$$\mathsf{CallStack}_{\mathsf{TFA}} = \mathsf{Frame}^{*}_{\mathsf{TFA}}$$

$$\mathsf{Conf}_{\mathsf{TFA}} = \mathsf{Heap} \times \mathsf{CallStack}_{\mathsf{TFA}}$$

The semantic reduction rules for the ordinary instructions are only changed in a trivial way to carry along the current transaction depth and annotate invoked

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ c}{P \vdash \langle H, \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \theta, m^{\theta_m}, pc+1, L, n :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{pop}\ \ n}{P \vdash \langle H, \langle \theta, m^{\theta_m}, pc, L, c :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \theta, m^{\theta_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ \ m_0 \\ S = v_1 :: \cdots :: v_{|m_0|} :: loc :: S_0 \qquad m_v = methodLookup(m_0, o.class) \\ o = H(loc) \wedge L' = [0 \mapsto loc, 1 \mapsto v_1, \ldots, |m_0| \mapsto v_{|m_0|}] \end{array}}{\begin{array}{r} P \vdash \langle H, \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \theta, m_v^{\theta}, 0, L', \epsilon\rangle :: \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{return}\ \ t}{\begin{array}{r} P \vdash \langle H, \langle \theta', m'^{\theta_{m'}}, pc', L', v :: S'\rangle :: \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \theta', m^{\theta_m}, pc+1, L, v :: S\rangle :: SF\rangle \end{array}}$$

Figure 5.1: Updated semantic reduction rules

methods. The updated rules are illustrated in Figure 5.1. The full semantics can be found in Appendix C.

The reduction rules for the instructions modelling the transaction API are shown in Figure 5.2. The rules implicitly encode the rules for well-formed transactions: an transaction can only be opened, using the `API.beginTransaction` instruction, if the current transaction depth is 0 and vice versa for the instructions `API.commitTransaction` and `API.abortTransaction`. Note that since Carmel Core does not include support for exceptions, the program is simply stuck when an error occurs.

To conclude the semantics the initial configurations are defined. No transactions are open in the initial configurations:

**Definition 5.1 (Initial Configurations).** *For a program, $P \in$ Program, the configuration $C \in$ Conf is an* initial configuration *if and only if $\sigma \in P.main$, $m_\sigma = \sigma.entry$, $C = \langle H, \langle 0, m_\sigma^0, 0, [0 \mapsto loc_\sigma], \epsilon\rangle :: \epsilon\rangle$ and $\forall \tau \in P.main : \exists loc_\tau : H(loc_\tau) = \tau$.*

### 5.1.2 Well-Formed Transactions

With the semantics in place a formal definition of well-formed transactions can be given:

**Definition 5.2 (Well-Formed Transactions).** *Let $C_0$ be an initial configuration of $P \in$ Program then the transactions of $P$ are said to be* well-formed *if and only if for all configurations $C_1 = \langle H, \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle$ such that $P \vdash C_0 \Longrightarrow_{TFA}^{*} C_1$ the following holds:*

1. $m.instructionAt(pc) = \texttt{API.beginTransaction} \Rightarrow \theta = 0$

$$\frac{m.instructionAt(pc) = \texttt{API.getTransactionDepth}}{P \vdash \langle H, \langle \theta, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \theta, m^{\theta_m}, pc+1, L, \theta :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.beginTransaction}}{P \vdash \langle H, \langle 0, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 1, m^{\theta_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.commitTransaction}}{P \vdash \langle H, \langle 1, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 0, m^{\theta_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.abortTransaction}}{P \vdash \langle H, \langle 1, m^{\theta_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 0, m^{\theta_m}, pc+1, L, S\rangle :: SF\rangle}$$

Figure 5.2: Semantics for transaction API

2. $m.instructionAt(pc) = \texttt{API.commitTransaction} \Rightarrow \theta = 1$

3. $m.instructionAt(pc) = \texttt{API.abortTransaction} \Rightarrow \theta = 1$

In essence this definition implies that the transaction instructions always succeed in a program with well-formed transactions and therefore the program does not end in a stuck configuration when executing such an instruction.

**Example 5.3.** *In Figure 5.3 an example Carmel method is shown that illustrates some of the techniques used to avoid breaking the well-formedness of transactions. The method, named* `atomicWrapper`*, is a "wrapper" method for the method invocation, in line 12, of the method named* `dosomething`*. The intention with the wrapper method is to ensure that the data updates performed in* `dosomething` *are always performed atomically, i.e., the wrapper must ensure that* `dosomething` *is always invoked inside an active transaction. The wrapper method, however, can not simply start a transaction since a transaction may already be active when the wrapper method is called. Therefore the wrapper must first check if a transaction is already active (lines 0–5) and if not, then it must start one (lines 6–8). Similarly, the wrapper method should only commit the transaction if it was opened by the wrapper (lines 13–15).*

*The wrapper method is a simplified Carmel version of a similar wrapper method found in the Demoney (see [Siv03b, MM02, Mar02]) demo-applet.*

It should be clear that the programming style exhibited in the above example is quite error prone and that it exacerbates the difficulty of verifying the safety and security of a program. The example also highlights some of the problems a program analysis of transactions must overcome: transactions started in one method may be committed or aborted in another method; the analysis must therefore be able to track inter-procedural control and transaction flow. It must also include a data flow component since the actual transaction depth is used to

```
void atomicWrapper(byte [], short)
{
  0: API.getTransactionDepth
  1: if s le 0 goto 4
  2: push s 1
  3: goto 5
  4: push s 0
  5: store s 3
  6: load s 3
  7: if s ne 0 goto 9
  8: API.beginTransaction
  9: load r 0
 10: load r 1
 11: load s 2
 12: invokevirtual dosomething(byte[], short)
 13: load s 3
 14: if s ne 0 goto 16
 15: API.commitTransaction
 16: return
}
```

Figure 5.3: Carmel implementation of a wrapper method

guard whether or not a transaction is started/ended. Consequently an analysis must combine control, data, and transaction flow components.

### 5.1.3 Abstract Domains

The transaction flow analysis specified in the next section is a combination of the control and data flow analyses with an analysis that conservatively approximates the transaction depth throughout a program. The data flow analysis is necessary to exploit the defensive programming style used by programmers to try to ensure that the transactions in a program are well-formed. Since 0 and 1 are the only possible transaction depths a simple instance of the data flow analysis is sufficient:

$$\overline{\mathsf{Num}}_{\mathsf{TFA}} = \overline{\mathsf{Num}}_1^{\top} = (\mathbb{Z} \times \{0,1\}) \cup \{\mathsf{INT}_N\}$$

where $\mathsf{INT}_N$ is the top value for the domain. This choice of abstract domain has the advantage that there is no need to annotate or instrument the numbers in the semantics in order to prove the analysis, cf. Section 5.1.5. The changed domain for abstract numbers is reflected in the domain for abstract values:

$$\overline{\mathsf{Val}}_{\mathsf{TFA}} = \overline{\mathsf{ObjRef}} + \overline{\mathsf{Num}}_{\mathsf{TFA}} \qquad \text{and} \qquad \widehat{\mathsf{Val}}_{\mathsf{TFA}} = \mathcal{P}(\overline{\mathsf{Num}}_{\mathsf{TFA}})$$

and for abstract objects:

$$\widehat{\mathsf{Object}}_{\mathsf{TFA}} = \mathsf{Field} \to \widehat{\mathsf{Val}}_{\mathsf{TFA}}$$

In order to use the analysis for verifying well-formedness of transactions in cases where a transaction is started in one method and ended in another, the invocations of methods in states with different transaction status must be separated in the analysis. To achieve this the analysis is made *context dependent* and the transaction depth is used as context. Transactions depths are modelled abstract as follows:

$$\widehat{\mathsf{TransDepth}} = \{0, 1\}$$

The context dependency is evident in the abstract domains for abstract local heaps:

$$\widehat{\mathsf{LocHeap}}_{\mathsf{TFA}} = \widehat{\mathsf{TransDepth}} \to \overline{\mathsf{Addr}} \to \mathbb{N}_0 \to \widehat{\mathsf{Val}}$$

abstract stacks:

$$\widehat{\mathsf{Stack}}_{\mathsf{TFA}} = \widehat{\mathsf{TransDepth}} \to \overline{\mathsf{Addr}} \to (\widehat{\mathsf{Val}}_{\mathsf{TFA}}{}^*)^\top$$

and abstract heaps:

$$\widehat{\mathsf{Heap}}_{\mathsf{TFA}} = \widehat{\mathsf{TransDepth}} \to \overline{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}_{\mathsf{TFA}}$$

Finally a *transaction cache* is needed to track the possible transaction depths an instruction may be executed in:

$$\widehat{\mathsf{Trans}} = \widehat{\mathsf{TransDepth}} \to \overline{\mathsf{Addr}} \to \mathcal{P}(\widehat{\mathsf{TransDepth}})$$

Intuitively $\delta' \in \hat{T}_\delta(m, pc)$ with $\hat{T}_\delta(m, pc) \in \widehat{\mathsf{Trans}}$ means that if method $m$ is invoked in a context with transaction depth $\delta$ then the instruction at address $(m, pc)$ may be executed with a transaction depth of $\delta'$. Note that $\delta$ is the *context* used to add precision to the analysis (by keeping invocations of a given method in different transaction depths separate) while $\delta'$ the analysis *result* for the given address in the given context. In Section 5.1.6 the $\hat{T}$ component is used to guarantee the well-formedness of transactions.

The above then leads to the following base domain for the analysis:

$$\widehat{\mathsf{Analysis}}_{\mathsf{TFA}} = \widehat{\mathsf{Heap}}_{\mathsf{TFA}} \times \widehat{\mathsf{LocHeap}}_{\mathsf{TFA}} \times \widehat{\mathsf{Stack}}_{\mathsf{TFA}} \times \widehat{\mathsf{Trans}}$$

A transaction flow analysis based on the above abstract domains is developed and specified in the next section.

### 5.1.4 Transaction Flow Logic

The Flow Logic judgements of the transaction flow analysis take the following form:

$$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{instr}$$

where $(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \in \widehat{\mathsf{Analysis}}_{\mathsf{TFA}}$. For legibility the context of an abstract domain is written as a subscript rather than a parameter, e.g., $\hat{S}(\delta)(m, pc)$ is written $\hat{S}_\delta(m, pc)$. Furthermore, the shorthand notation $\hat{T}_{\{0,1\}}(m, pc)$ is taken to mean $\hat{T}_0(m, pc) \cup \hat{T}_1(m, pc)$.

Since the analysis is context dependent and the context is the transaction depth in which the current method was invoked, the Flow Logic specification for all instructions start by making sure that all possible contexts (in which the current method was invoked) are taken into account; formally this is done, for each instruction, by quantifying over all the possible contexts that instruction can be executed in and then perform the analysis in all those contexts. This is implemented in the analysis by prefixing all the specifications with the following:

$$\forall \delta \in \hat{T}_{\{0,1\}}(m, 0)\colon \hat{T}_{\delta}(m, pc) \neq \emptyset \Rightarrow \ldots$$

The condition that $\hat{T}_{\delta}(m, pc) \neq \emptyset$ ensures that instructions are not analysed in contexts that are *known* to be unreachable.

Below the Flow Logic specifications for a few instructions are discussed in detail. The full specification can be found in Appendix C.

### Arithmetic and Conditionals

Since transactions depths are typically only stored and never used in actual computations a very simple constant tracking data flow analysis was chosen. The data flow analysis is unable to track constants through computations. This is most evident in the specification for `numop`, see Figure 5.4, since it always results in the top value, $\mathsf{INT}_N$. The specification is also updated to copy forward the current abstract transaction depth:

$$\hat{T}_{\delta}(m, pc) \subseteq \hat{T}_{\delta}(m, pc + 1)$$

Conditionals are simply analysed as a context dependent version of the data flow analysis using the same abstract condition predicate, $\mathsf{COND}_{\mathsf{Num}}$, as specified in Section 4.1.5.

### Method Invocation

The specification for `invokevirtual` follows the pattern laid from previous analyses. The main difference being that the analysis is extended to take contexts into account. Further changes are related to ensuring that the abstract transaction depth is handled correctly. First the current abstract context is added as a "base" context for the invoked method:

$$\forall \delta' \in \hat{T}_{\delta}(m, pc)\colon \{\delta'\} \subseteq \hat{T}_{\delta'}(m_v, 0)$$

Similarly the abstract transaction depth when the invoked method returns must be copied back to the invoking method and used as the current depth:

$$\hat{T}_{\delta'}(m_v, \mathsf{END}) \subseteq \hat{T}_{\delta}(m, pc + 1)$$

See Figure 5.5 for the full specification of `invokevirtual` and `return`.

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{numop}\ \ t\ \ binop \\
\quad \text{iff} \quad \forall \delta \in \hat{T}_{\{0,1\}}(m,0) \colon \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow \\
\qquad A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m,pc) : \\
\qquad \{\mathsf{INT}_N\} :: X \sqsubseteq \hat{S}_\delta(m,pc+1) \\
\qquad \hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1) \\
\qquad \hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0 \\
\quad \text{iff} \quad \forall \delta \in \hat{T}_{\{0,1\}}(m,0) \colon \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow \\
\qquad A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m,pc) : \\
\qquad \mathsf{COND}_{\mathsf{Num}}(cmp, A_1, A_2) \Rightarrow \\
\qquad\quad \hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc_0) \\
\qquad\quad \hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc_0) \\
\qquad\quad \hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc_0) \\
\qquad \mathsf{COND}_{\mathsf{Num}}(\neg cmp, A_1, A_2) \Rightarrow \\
\qquad\quad \hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1) \\
\qquad\quad \hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1) \\
\qquad\quad \hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)
\end{array}
$$

Figure 5.4: Flow Logic specification arithmetic and conditionals

**Transactions**

The Flow Logic specification for the instructions concerned with transactions can be seen in full in Figure 5.6.

The analysis of `API.getTransactionDepth` simply specifies that the current (abstract) transaction depth is pushed on top of the stack:

$$\forall \delta' \in \hat{T}_\delta(m,pc) \colon \{\delta'\} :: \hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$$

As for other instructions the local heap and current transaction depths are copied forward.

The `API.beginTransaction` is analysed in a similar way except that here the current transaction depth is not copied forward, instead it is changed to 1 to indicate that a transaction is now active:

$$\{1\} \subseteq \hat{T}_\delta(m,pc+1)$$

For `API.commitTransaction` and `API.abortTransaction` the current transaction depth is, of course, changed to 0 instead of 1. This depends on the fact that, unless the transaction are not well-formed, the transaction depth is *always* either 0 or 1. In particular: after a transaction has been opened or closed transaction depth will always be either 1 or 0 respectively. To accommodate deeper nesting it would be necessary to take the current transaction depth into account and then add (or subtract) one when opening (or closing) a transaction.

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{invokevirtual}\ m_0 \\
\quad \text{iff} \quad \forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow \\
\qquad A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}_\delta(m,pc) : \\
\qquad \forall (\mathsf{Ref}\ \sigma) \in B : \\
\qquad\quad m_v \triangleleft methodLookup(m_0,\sigma) : \\
\qquad\quad \forall \delta' \in \hat{T}_\delta(m,pc) : \\
\qquad\qquad \{\delta'\} \subseteq \hat{T}_{\delta'}(m_v,0) \\
\qquad\qquad \{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}_{\delta'}(m_v,0)[0..|m_0|] \\
\qquad\qquad \hat{T}_{\delta'}(m_v,\mathsf{END}) \subseteq \hat{T}_\delta(m,pc+1) \\
\qquad\qquad m.returnType \neq \texttt{void} \Rightarrow \\
\qquad\qquad\quad A :: Y \triangleleft \hat{S}_{\delta'}(m_v,\mathsf{END}) : \\
\qquad\qquad\qquad A :: X \sqsubseteq \hat{S}_\delta(m,pc+1) \\
\qquad\qquad m.returnType = \texttt{void} \Rightarrow \\
\qquad\qquad\quad X \sqsubseteq \hat{S}_\delta(m,pc+1) \\
\qquad \hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{return}\ t \\
\quad \text{iff} \quad \forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow \\
\qquad \hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,\mathsf{END}) \\
\qquad A :: X \triangleleft \hat{S}_\delta(m,pc) : \\
\qquad\quad A :: \epsilon \sqsubseteq \hat{S}_\delta(m_0,\mathsf{END})
\end{array}
$$

Figure 5.5: Flow Logic specification for method support

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.getTransactionDepth}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$\forall \delta' \in \hat{T}_\delta(m,pc) : \{\delta'\} :: \hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.beginTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$\{1\} \subseteq \hat{T}_\delta(m,pc+1)$
$\hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.commitTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$\{0\} \subseteq \hat{T}_\delta(m,pc+1)$
$\hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.abortTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$\{0\} \subseteq \hat{T}_\delta(m,pc+1)$
$\hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$

Figure 5.6: Flow Logic specification for transaction API

**Programs**

The analysis of a full program can then be specified as follows:

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} P \quad \text{iff} \\
\quad \forall (m,pc) \in P.addresses: \\
\quad\quad m.instructionAt(pc) = \texttt{instr} \Rightarrow (\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{instr} \\
\quad \forall \sigma \in P.classes: m_\sigma = \sigma.entry \Rightarrow \\
\quad\quad (\mathsf{Ref}\ \sigma) \in \hat{L}_0(m_\sigma,0) \wedge \{0\} \subseteq \hat{T}_0(m_\sigma,0)
\end{array}
$$

### 5.1.5 Semantic Correctness

Proving semantic correctness follows the same approach as the previous analyses: first the representation functions and correctness relations are defined and then a subject reduction property is established.

Because of the simple domain for abstract numbers it is sufficient to represent numbers as themselves:

$$\beta_{\mathsf{TFA,Num}}(n) = \{n\}$$

This is possible because the analysis immediately maps any numerical operation to the top value, $\mathsf{INT}_N$. The representation function for values is changed accordingly:

$$\beta^H_{\mathsf{TFA,Val}}(v) = \begin{cases} \beta_{\mathsf{TFA,Num}}(v) & \text{if } v \in \mathsf{Num_{TFA}} \\ \beta^H_{\mathsf{Ref}}(v) & \text{if } v \in \mathsf{Ref} \end{cases}$$

as are the functions for stacks:

$$\beta^H_{\mathsf{TFA,Stack}}(v_1 :: \cdots :: v_n) = \beta^H_{\mathsf{TFA,Val}}(v_1) :: \cdots :: \beta^H_{\mathsf{TFA,Val}}(v_n)$$

local heaps:

$$\beta^H_{\mathsf{TFA,LocHeap}}(L)(x) = \beta^H_{\mathsf{TFA,Val}}(L(x))$$

and objects:

$$\beta^H_{\mathsf{TFA,Object}}(o)(f) = \beta^H_{\mathsf{TFA,Val}}(o.f)$$

and finally the representation function for heaps:

$$\beta_{\mathsf{TFA,Heap}}(H)(\mathsf{Ref}\ \sigma) = \bigsqcup_{\substack{loc \in \mathrm{dom}(H) \\ \beta^H_{\mathsf{TFA,Val}}(loc) = (\mathsf{Ref}\ \sigma)}} \beta_{\mathsf{TFA,Object}}(H(loc))$$

The remaining representation functions are the same as those defined in Section 3.3. With the representation functions for the basic domains in place, the correctness relations can be defined. First for stack frames:

$$
\begin{array}{lll}
\langle \theta, m^{\theta_m}, pc, L, S\rangle\ \mathcal{R}^H_{\mathsf{TFA,Frame}}\ (\hat{L},\hat{S},\hat{T}) & \text{iff} & \theta_m \in \hat{T}_{\{0,1\}}(m,0) \wedge \\
 & & \theta \in \hat{T}_{\theta_m}(m,pc) \wedge \\
 & & \beta^H_{\mathsf{TFA,LocHeap}}(L) \sqsubseteq \hat{L}_{\theta_m}(m,pc) \wedge \\
 & & \beta^H_{\mathsf{TFA,Stack}}(S) \sqsubseteq \hat{S}_{\theta_m}(m,pc)
\end{array}
$$

The above correctness relation essentially formalises that the stack frame should be (abstractly) represented in the analysis in such a way that the context (transaction depth) in which the current method, $m$, was called, $\theta_m$, is used as a context for the analysis, e.g., as in $\hat{S}_\delta(m, pc)$, and also that it is represented correctly in the analysis: $\theta_m \in \hat{T}_{\{0,1\}}(m, 0)$. This is easily extended to cover call stacks:

$$F_1 :: \cdots :: F_n \; \mathcal{R}^H_{\mathsf{TFA,CallStack}} \; (\hat{L}, \hat{S}, \hat{T}) \quad \text{iff} \quad \forall i : F_i \; \mathcal{R}^H_{\mathsf{TFA,Frame}} \; (\hat{L}, \hat{S}, \hat{T})$$

This leads to the following unsurprising correctness relation for semantic configurations:

$$\langle H, SF \rangle \; \mathcal{R}^H_{\mathsf{TFA,Conf}} \; (\hat{H}, \hat{L}, \hat{S}, \hat{T}) \quad \text{iff} \quad \begin{array}{l} \beta_{\mathsf{TFA,Heap}}(H) \sqsubseteq \hat{H} \wedge \\ SF \; \mathcal{R}^H_{\mathsf{TFA,CallStack}} \; (\hat{L}, \hat{S}, \hat{T}) \end{array}$$

which is then used to state and prove the semantic correctness of the transaction flow analysis:

**Theorem 5.4 (Soundness).** *If* $P \in \mathsf{Program}$ *such that* $(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{TFA} P$ *and* $C_0$ *an initial configuration of* $P$ *with* $P \vdash C_0 \Longrightarrow^*_{TFA} C_1$ *and* $P \vdash C_1 \Longrightarrow_{TFA} C_2$ *then*

$$C_1 \; \mathcal{R}_{\mathsf{TFA,Conf}} \; (\hat{H}, \hat{L}, \hat{S}, \hat{T}) \;\Rightarrow\; C_2 \; \mathcal{R}_{\mathsf{TFA,Conf}} \; (\hat{H}, \hat{L}, \hat{S}, \hat{T})$$

*Proof.* By extension of the proof for Theorem 3.14. Below the proof case for `API.beginTransaction` is detailed for illustration.

*Case* `API.beginTransaction`*:* By assumption:

$$\frac{m.instructionAt(pc) = \texttt{API.beginTransaction}}{P \vdash \langle H, \langle 0, m^{\theta_m}, pc, L, S \rangle :: SF \rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 1, m^{\theta_m}, pc + 1, L, S \rangle :: SF \rangle}$$

and

$$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.beginTransaction} \tag{5.1}$$

and

$$\langle H, \langle 0, m^{\theta_m}, pc, L, S \rangle :: SF \rangle \; \mathcal{R}_{\mathsf{TFA,Conf}} \; (\hat{H}, \hat{L}, \hat{S}, \hat{T}) \tag{5.2}$$

Equation (5.2) implies that

$$\begin{array}{rcll} \beta(H) & \sqsubseteq & \hat{H} & (5.3) \\ \beta^H(L) & \sqsubseteq & \hat{L}_{\theta_m}(m, pc) & (5.4) \\ \beta^H(S) & \sqsubseteq & \hat{S}_{\theta_m}(m, pc) & (5.5) \\ 0 & \in & \hat{T}_{\theta_m}(m, pc) & (5.6) \\ \theta_m & \in & \hat{T}_{\{0,1\}}(m, 0) & (5.7) \end{array}$$

Thus since $\theta_m \in \hat{T}_{\{0,1\}}$ and $0 \in \hat{T}_{\theta_m}(m, pc)$ it follows from the definition of the analysis and equation (5.1) that

$$\{1\} \subseteq \hat{T}_{\theta_m}(m, pc+1) \tag{5.8}$$
$$\hat{S}_{\theta_m}(m, pc) \sqsubseteq \hat{S}_{\theta_m}(m, pc+1) \tag{5.9}$$
$$\hat{L}_{\theta_m}(m, pc) \sqsubseteq \hat{L}_{\theta_m}(m, pc+1) \tag{5.10}$$

From which the case follows.

The remaining cases are similar. ∎

Note that Lemma 3.11 can be used without modification since neither the transaction instruction nor the addition of transaction depths to the stack frames influence the call stack.

### 5.1.6 Static Well-Formedness

Having defined the transaction flow analysis and shown that it correctly approximates the semantics, it is now possible to exploit the subject reduction property for statically verifying that the transactions in a program are indeed well-formed. To this end the notion of *static well-formedness* is defined:

**Definition 5.5 (Static Well-Formedness).** *A program $P \in$ Program is said to have* statically well-formed *transactions if and only if $(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{TFA} P$ and the following holds:*

1. $m.instructionAt(pc) = \texttt{API.beginTransaction} \Rightarrow \hat{T}_{\{0,1\}}(m, pc) \subseteq \{0\}$
2. $m.instructionAt(pc) = \texttt{API.commitTransaction} \Rightarrow \hat{T}_{\{0,1\}}(m, pc) \subseteq \{1\}$
3. $m.instructionAt(pc) = \texttt{API.abortTransaction} \Rightarrow \hat{T}_{\{0,1\}}(m, pc) \subseteq \{1\}$

Intuitively, if $m.instructionAt(pc) = \texttt{API.beginTransaction}$ for some $(m, pc)$ and $\hat{T}_{\{0,1\}}(m, pc) \subseteq \{0\}$ then no matter what context method $m$ was invoked in (indicated by the $\{0, 1\}$ subscript to $\hat{T}$) the instruction executed at address $(m, pc)$ (the opening of a transaction) can *only* be executed in a context where the transaction depth is 0 (since $\hat{T}_{\{0,1\}} \subseteq \{0\}$) which is exactly what is required for well-formedness, cf. Definition 5.2. This is formalised in the following theorem:

**Theorem 5.6.** *Let $P \in$ Program such that $(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{TFA} P$ and let $P$ have statically well-formed transactions, then $P$ has well-formed transactions.*

*Proof.* Follows immediately from Theorem 5.4 and definitions 5.2 and 5.5. ∎

```
void atomicWrapper(byte [], short)
{
  0: API.getTransactionDepth                    /* {0} {1} */
  1: if s le 0 goto 4                           /* {0} {1} */
  2: push s 1                                   /* {}  {1} */
  3: goto 5                                     /* {}  {1} */
  4: push s 0                                   /* {0} {}  */
  5: store s 3                                  /* {0} {1} */
  6: load s 3                                   /* {0} {1} */
  7: if s ne 0 goto 9                           /* {0} {1} */
  8: API.beginTransaction                       /* {0} {}  */
  9: load r 0                                   /* {1} {1} */
 10: load r 1                                   /* {1} {1} */
 11: load s 2                                   /* {1} {1} */
 12: invokevirtual dosomething(byte[], short)   /* {1} {1} */
 13: load s 3                                   /* {1} {1} */
 14: if s ne 0 goto 16                          /* {1} {1} */
 15: API.commitTransaction                      /* {1} {}  */
 16: return                                     /* {0} {1} */
}
```

Figure 5.7: Carmel implementation of a wrapper method with transaction flow analysis results

This theorem formally shows that the transaction flow analysis can be used to statically verify that a given program has well-formed transactions and therefore that the program does not give rise to transaction-related exceptions.

**Example 5.7.** *In Figure 5.7 the example program from Figure 5.3 is repeated along with the results of the transaction flow analysis. For convenience the results are shown as in-lined comments of the form* `/* X Y */` *where* `X` *and* `Y` *corresponds to* $\hat{T}_0$ *and* $\hat{T}_1$ *respectively.*

*The first thing to note is that the conditions for statically well-formed transactions set out in Definition 5.5 are fulfilled in the wrapper method (lines 8 and 15). The second thing to note is that the* `dosomething` *method is always invoked with a transaction depth of 1 and thus it is always invoked inside a transaction and therefore atomicity of the updates performed in* `dosomething` *is guaranteed.*

### 5.1.7 Implementation

A prototype of the transaction flow analysis has been implemented by systematically extending implementation of the control flow analysis (Section 3.4) with context information and a transaction component. The prototype was used to produce the analysis result shown in Example 5.7.

## 5.2 Secure Information Flow

The use of smart cards for authentication and high-security applications often requires highly confidential information to be stored on the card, e.g., personal information or private cryptographic keys. The on-card programs that manipulate confidential data must be designed and implemented very carefully in order to ensure that secret information is not leaked. Preventing information leaks in computer applications and information systems is a classic problem that has been studied extensively, at least in the context of military systems. One of the earliest security models formalising the notions of security classifications and confidentiality is the *multi-level security* (MLS) model (also known as the Bell/LaPadula model), cf. [BL73a, BL73b], where IT systems are seen as collections of *subjects*, e.g., users and processes, that actively access and modify *objects*, e.g., files and records. Both subjects and objects are assigned security classifications and security is then guaranteed if no subject can read an object with a higher security classification than itself, this is called the "no-read-up" property, and if no subject can write to an object with a lower security classification than itself, called the "no-write-down" property. The MLS model provided the formal basis for the "Trusted Computer Security Evaluation Criteria" (better known as the "orange book") cf. [DoD85], a standard for evaluating the security of IT systems later succeeded by the Common Criteria [CC99]. While the MLS model is conceptually simple and easy to apply, the notion of security guaranteed by the model depends very much on the interpretation and specific semantics of the underlying operations, sometimes resulting in counter-intuitive notions of security. This is demonstrated in [McL85] where it is shown how a patently insecure system can be proved "secure" in the MLS model.

This led to the development of more abstract models such as *strong dependency* in [Coh77, Coh78] and especially *non-interference* in [GM82, GM84]. The notion of *non-interference* was suggested as an abstract security policy. Rather than specifying how access to data should be controlled, as in the MLS model, non-interference takes an end-to-end view of security and only specifies what information may or may not be observed from various points in the system. Implementations must then be shown to incorporate adequate security measures, e.g., access controls, to guarantee non-interference. A major advantage of non-interference and other information flow based security policies is that all information channels in the model, including *covert channels* cf. [Lam73], are covered by the policy. For access control based models, e.g. MLS, such channels often have to be modelled explicitly in order to be covered by the policies and security guarantees. In [Mil87] it is shown that non-interference provides near perfect secrecy in the sense of Shannon's information theory [Sha48]. The papers [McL94, Lan81, McL90] provide good overviews and introductions to formal information security models. See [Bis03] for a recent and comprehensive overview.

Originally the security models mentioned above were mostly developed to formulate and prove the security of operating systems but quickly found use in language-based security. An early example is [DD77] where an information

flow analysis, based on the lattice model [Den76], is defined for a simple while-language and used to verify that programs do not leak secret information. Even though the analysis is specified formally the proof of correctness is informal and is, in essence, only a proof that programs manipulate data in accordance with the MLS model, i.e., the no-read-up and no-write-down properties. This was rectified in [VSI96] where a type-system was developed, based on the analysis of [DD77], to enforce the security of information flow in a simple while-language by proving that well-typed programs have the non-interference property. Since then research into language-based information flow security has been prolific and found widespread use; see [SM03] for a comprehensive overview.

In this section a notion of non-interference for Carmel Core programs is defined. Moreover an *information flow analysis* is developed to automatically verify that a given Carmel Core program has the non-interference property and thus can not leak secret information. Only comparatively little research has been made into secure information flow for low-level languages such as the Java (Card) bytecode language: [BBR04, GS05, KS02, ABF03]. The notion of non-interference defined for Carmel Core below is inspired by the work in [FSBJ97, SBCJ97] for secure information flow in object-oriented systems.

To simplify the proofs in the following sections it is necessary to slightly restrict the class of programs considered:

**Remark 5.8 (Recursion).** *In this section only programs that do not contain recursive method invocations are considered.*

The lack of support for recursive methods is not a problem in practice since applets rarely, if ever, use recursion due to the extra complexity and work required to argue or prove that recursion is used correctly and safely.

### 5.2.1 Non-Interference for Carmel Core

The notion of security chosen here is based on the observation that the information that must be protected on a smart card typically resides in certain instance fields of objects in memory. This leads to a definition of non-interference that makes it possible to detect and/or prevent information leaks from instance fields by assigning security classifications only to instance fields. Furthermore, the way objects are used in the current generation of smart card applets it is too inflexible to require that entire objects or classes are classified as high security even if they contain fields that are high security, e.g., as in [ABF03]. This leads to a definition of security that is loosely inspired by [FSBJ97, SBCJ97] but allowing for more fine-grained security policies since only fields, not entire objects, have security classifications. Other approaches define security as a (lack of) dependency between a methods parameters (input) and its return value (output), cf. [KS02, GS05].

For simplicity only two security levels are considered: high ($\mathsf{H}$) and low ($\mathsf{L}$) with $\mathsf{L} \sqsubseteq \mathsf{H}$:

$$\mathsf{Security} = (\{\mathsf{L}, \mathsf{H}\}, \sqsubseteq)$$

The main technical results in the following can easily be adapted to allow for arbitrary lattices of security classifications. A security policy is now defined as an assignment of security levels to all instance fields:

**Definition 5.9 (Security policy).** *A* security policy *is a total function,* $level : \mathsf{Field} \rightarrow \mathsf{Security}$*, that assigns a security level to instance fields.*

Dynamic allocation of memory on the heap and thus of locations, i.e., through the `new`-instruction, and the fact that the dynamically allocated locations are first class values occurring in programs makes it more complicated to define non-interference since it can only be defined up to isomorphism on locations. The following relation compares values in a program up to a mapping $\pi$. The map is required to be bijective (on the subset of $\mathsf{Loc}$ on which it is defined); in the definition of heap equivalence (Definition 5.11) the map must be an isomorphism on the locations that point to objects containing fields with a low-security classification:

**Definition 5.10 ($\pi$-equivalence).** *Let* $v_1, v_2 \in \mathsf{Val}$ *and* $\pi : \mathsf{Loc} \rightarrow \mathsf{Loc}$ *be a bijective partial map and define*

$$v_1 \equiv_\pi v_2 \quad \textit{iff} \quad \begin{cases} v_1 = v_2 & \textit{if } v_1, v_2 \notin \mathsf{Loc} \\ v_2 = \pi(v_1) & \textit{if } v_1 \in \mathrm{dom}(\pi) \\ v_1 = \pi^{-1}(v_2) & \textit{if } v_2 \in \mathrm{codom}(\pi) \\ \textit{true} & \textit{if } v_1 \in \mathsf{Loc} \setminus \mathrm{dom}(\pi), v_2 \in \mathsf{Loc} \setminus \mathrm{codom}(\pi) \end{cases}$$

As already mentioned, the kind of non-interference of interest here must be able to prevent leaks from high-security instance fields to low-security instance fields. Local variables and stack contents are of no concern here since they are only used temporarily for storing secret information. Thus security, as defined here, is only concerned with the contents if the heap. The following definition formalises that two heaps are considered to be equivalent, as seen from a security perspective, when all fields that have a low security classification are equivalent (modulo the isomorphism on low heap locations):

**Definition 5.11 (Heap-equivalence).** *Let* $H_1, H_2 \in \mathsf{Heap}$*, then* $H_1 \approx_{\mathsf{L}} H_2$ *if and only if there exists a bijective partial map,* $\pi : \mathsf{Loc} \rightarrow \mathsf{Loc}$*, such that*

$$\forall loc_1 \in \mathrm{dom}(H_1) : \forall f \in H_1(loc_1).\mathit{fields} : f.\mathit{level} \sqsubseteq \mathsf{L} \Rightarrow \\ H_1(loc_1).\mathit{class} = H_2(\pi(loc_1)).\mathit{class} \wedge H_1(loc_1).f \equiv_\pi H_2(\pi(loc_1)).f$$

*and*

$$\forall loc_2 \in \mathrm{dom}(H_2) : \forall f \in H_2(loc_2).\mathit{fields} : f.\mathit{level} \sqsubseteq \mathsf{L} \Rightarrow \\ H_1(\pi^{-1}(loc_2)).\mathit{class} = H_2(loc_2).\mathit{class} \wedge H_1(\pi^{-1}(loc_2)).f \equiv_\pi H_2(loc_2).f$$

A mapping, $\pi$, that fulfils the above requirements is called a *low-isomorphism* on locations. Note that $\pi$ is not defined for all locations, only those that point to objects that contain at least one field of a low security classification. Thus any object that is composed entirely of high-security fields is "invisible" to a low security observer. It is now possible to define non-interference for Carmel:

**Definition 5.12 (Non-Interference).** *Let* $P \in \mathsf{Program}$, $H_1, H_2 \in \mathsf{Heap}$ *and let* $\langle H_i, \langle m_i, 0, L_i, \epsilon \rangle\rangle$ *for* $i = 1, 2$ *be initial configurations for* $P$ *such that*

$$P \vdash \langle H_1, \langle m_1, 0, L_1, \epsilon \rangle\rangle \Longrightarrow^* \langle H_1', \langle \mathsf{Ret}\ v_1 \rangle\rangle$$

*and*

$$P \vdash \langle H_2, \langle m_2, 0, L_2, \epsilon \rangle\rangle \Longrightarrow^* \langle H_2', \langle \mathsf{Ret}\ v_2 \rangle\rangle$$

*then* $P$ *is said to be* non-interfering *if and only if*

$$H_1 \approx_{\mathsf{L}} H_2 \;\Rightarrow\; H_1' \approx_{\mathsf{L}} H_2'$$

Intuitively this interpretation of non-interference states that if a given program is started in two different initial configurations with equivalent heaps, then the program is non-interfering if both executions terminate and the heaps in the final configurations are equivalent. This guarantees that no high-security field could have influenced any low-security field. Note that only the heap is needed to achieve the (informal) notion of security discussed in the previously in this section.

The definition of non-interference has a number of noteworthy implications. First, it only applies to terminating programs and thus cannot prevent information leaks through termination (or timing) behaviour. See [KJJ99, Koc96] for actual attacks exploiting timing behaviour and power-consumption. In [Aga00a] a program transformation that can eliminate such timing leaks is discussed; an implementation of this program transformation for Java bytecode is detailed in [Aga00b]. The second thing to note is the definition of non-interference, and thus security, could be extended to also cover return values and thus implement an aspect of input/output non-interference as well as heap-equivalence.

### 5.2.2 Information Flow Analysis

As noted in the preceding section it is quite hard to manually inspect and prove that a given program has the non-interference property. In the following an *information flow analysis* is defined that can automatically verify that a Carmel Core program is non-interfering. Intuitively the analysis must track information from high-security fields and make sure that such information is not stored in a low-security field. Information transfers of this kind are called *direct flows*. However, information can also be leaked through *implicit flows* where it is less obvious that there even is an information transfer. In Carmel Core the two major sources of implicit flows are conditionals and the use of references. A conditional can give rise to an implicit flow if the branching of the conditional depends on high-security information. Figure 5.8 shows a code snippet exemplifying an implicit flow through a conditional. Assuming that $f.level = \mathsf{H}$ and contains a numerical value, then the top of the stack will contain a zero if $f$ has the value zero and a one otherwise. In that way information about the values of the field $f$ can be leaked in small bits and conceivably be stored in a low-security field. The use of object references for instance fields and method

```
0: load r 0
1: getfield f
2: if eq 0 goto 5
3: push s 1
4: goto 6
5: push s 0
6: ...
```

Figure 5.8: Example of implicit flow

invocations is also a source of implicit flows since different fields or methods would be accessed/invoked depending on the specific object reference used thus leaking information about the object reference itself.

One of the main problems for an information flow analysis to overcome is to take implicit flows into account and ensure that they are handled correctly. The information flow analysis developed below incorporates a special component specifically to track the implicit flow of a program. However, there is another problem related to the implicit flows: conditionals in Carmel, and other low level languages, are basically conditional jumps and, in contrast to higher level languages, there is no program structure to indicate or even suggest the scope of a conditional statement. In order to recover (some of) that structure the analysis computes the *post-dominators* or forward dominators for all program points; such post-dominators represent program points where every terminating execution from the corresponding conditional *must* pass through regardless of the branch taken. In other words: the post-dominators are program points that do not depend on the actual result of the conditional and consequently no information, implicit or explicit, can be leaked from the conditional to the post-dominators. This technique has long been known and used in compilers, e.g., to construct the control dependence graph necessary for certain program optimisations, cf. [App98, ASU85]. A similar approach is taken in [KS02, ABF03]. For a configuration $\langle H, \langle m, pc, L, S\rangle :: SF\rangle$ let $C.address = (m, pc)$. The post-dominator can then be formally defined in the current setting as follows:

**Definition 5.13 (Post-dominator).** *For $P \in \mathsf{Program}$ the program counter $pc$ is a* post-dominator *for $(m_1, pc_1)$, written $(m_1, pc_1) \curvearrowright pc_1'$, if for all reduction sequences with $C_1.address = (m_1, pc_1)$ and of the form:*

$$P \vdash C_0 \Longrightarrow^* C_1 \Longrightarrow \cdots \Longrightarrow C_n \Longrightarrow \langle H, \langle \mathsf{Ret}\ v\rangle\rangle$$

*there exists an $i \in \{2, \ldots, n\}$ such that $C_i.address = (m_1, pc_1')$.*

which leads to the definition of immediate post-dominators:

**Definition 5.14 (Immediate post-dominator).** *A program counter $pc$ is an* immediate post-dominator *for $(m_1, pc_1)$, written $(m_1, pc_1) \curvearrowright_i pc$, if and only if $(m_1, pc_1) \curvearrowright pc$ and there does not exist a $pc'$ such that $(m_1, pc_1) \curvearrowright pc'$ and $(m_1, pc') \curvearrowright pc$.*

Note that for simplicity only post-dominators in the same method as the dominated address are considered. This makes the analysis and proofs simpler at the cost of losing precision in the analysis.

Since the post-dominators are used directly in the analysis to handle implicit flows they must be represented explicitly in the Flow Logic specification. To this end the following domain is defined:

$$\mathsf{Dominators} = \overline{\mathsf{Addr}} \to \mathcal{P}(\mathsf{PC})$$

with the intended meaning that if $(m_1, pc_1) \curvearrowright pc_1'$ then $pc_1' \in DOM(m_1, pc_1)$.

Several efficient algorithms exist for computing the set of post-dominators and the corresponding immediate post-dominators, see [App98] for details. Here a less efficient but more readable alternative is chosen, based on the following (abstract) equations that define the set of post-dominators for a given control flow graph:

$$\begin{array}{rcl} DOM(m, \mathsf{END}) & = & \{\mathsf{END}_m\} \\ DOM(m, pc) & = & \{pc\} \cup \bigcap_{pc' \in succ(pc)} DOM(m, pc') \end{array}$$

where *succ* is the set of successor nodes in the corresponding control flow graph. These equations can then be specialised to the structure of a Carmel Core program:

**Definition 5.15.** *Let $P \in \mathsf{Program}$ and $DOM \in \mathsf{Dominators}$, then $DOM$ is the set of post-dominators for $P$, written $DOM(P)$, if and only if*

$$\forall m \in P.methods : DOM(m, pc) = \begin{cases} \{pc\} \cup (DOM(m, pc+1) \cap DOM(m, pc_0)) & \\ \qquad\qquad \textit{if } m.instructionAt(pc) = \texttt{if } t \textit{ cmp } \texttt{goto } pc_0 \\ \{\mathsf{END}_m\} & \textit{if } m.instructionAt(pc) = \texttt{return} \\ \{pc\} \cup DOM(m, pc+1) & \textit{otherwise} \end{cases}$$

Since most of the instructions do not change the control flow of a program the corresponding post-dominator is quite simple. Only the `if` and `return` instructions have more complex rules; even for the `invokevirtual`-instruction a trivial rule for post-dominators is sufficient since the notion of post-dominator defined here is always local to a method. In Section 5.2.4 a few of the rules are discussed in more detail in conjunction with the Flow Logic specification and in Section 5.2.5 the soundness of the above definition is proved (Lemma 5.16).

Given the dominator sets it is easy to determine the immediate dominator for a given address. First define

$$DOM^2(m, pc) = \{pc'' \mid pc'' \in DOM(m, pc'), pc' \in DOM(m, pc) \setminus \{pc\}\}$$

then the immediate post-dominator for $(m_1, pc_1)$ can be calculated as follows:

$$(DOM(m_1, pc_1) \setminus \{pc_1\}) \setminus DOM^2(m_1, pc_1)$$

Note that every node except the exit node has exactly one immediate post-dominator, cf. [App98, pages 407–409], and therefore the above formula is well-defined.

### 5.2.3 Abstract Domains

The abstract domains for the information flow analysis are mainly extensions of the abstract domains for the control flow analysis with security information. In addition a abstract domains are needed to compute the post-dominators, as discussed in the previous section, and to track the implicit flow.

First the abstract stack is extended to hold a security level for each stack position:

$$\widehat{\mathsf{Stack}}_{\mathsf{IFA}} = \overline{\mathsf{Addr}} \to ((\widehat{\mathsf{Val}} \times \mathsf{Security})^*)^\top$$

and similar changes are made to the local heap:

$$\widehat{\mathsf{LocHeap}}_{\mathsf{IFA}} = \overline{\mathsf{Addr}} \to \mathbb{N}_0 \to (\widehat{\mathsf{Val}} \times \mathsf{Security})$$

and for the instance fields of objects:

$$\widehat{\mathsf{Object}}_{\mathsf{IFA}} = \mathsf{Field} \to (\widehat{\mathsf{Val}} \times \mathsf{Security})$$

The domain for abstract heaps must then be updated to reflect the changes:

$$\widehat{\mathsf{Heap}}_{\mathsf{IFA}} = \overline{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}_{\mathsf{IFA}}$$

Finally, tracking implicit flow requires keeping track of the security label of the implicit flow and also the origin of the implicit flow, i.e., the program point of the conditional or method invocation that gave rise to the implicit flow:

$$\widehat{\mathsf{Implicit}} = \overline{\mathsf{Addr}} \to \mathcal{P}(\mathsf{Security} \times \overline{\mathsf{Addr}})$$

The least upper bound of the security levels of the possible implicit flows at an address, i.e., $\sqcup\{\ell' \mid (\ell', (m', pc'')) \in \hat{C}(m, pc)\}$, is called the *security context* of that address and is written $\sqcup\, \hat{C}(m, pc)$.

Implicit flows originating at program point $pc$ must be propagated throughout the program until a post-dominator for $pc$ is encountered. This is formalised as follows for $\hat{C}_1, \hat{C}_2 \in \widehat{\mathsf{Implicit}}$ and $DOM \in \mathsf{Dominators}$:

$$\begin{array}{l}\hat{C}_1(m_1, pc_1) \sqsubseteq_{DOM} \hat{C}_2(m_2, pc_2) \quad \text{iff} \\ \quad \{(\ell, (m, pc)) \in \hat{C}_1(m_1, pc_1) \mid m_2 \neq m \vee pc_2 \notin DOM(m, pc)\} \subseteq \hat{C}_2(m_2, pc_2)\end{array}$$

Note that implicit flows can only be removed in the method in which they are added. This results in an analysis where any implicit flows that are present when a new method is invoked will be copied into the invoked method as implicit flows for that method that can *not* be removed.

The domain for the information flow analysis can then be defined as:

$$\widehat{\mathsf{Analysis}}_{\mathsf{IFA}} = \widehat{\mathsf{Heap}}_{\mathsf{IFA}} \times \widehat{\mathsf{LocHeap}}_{\mathsf{IFA}} \times \widehat{\mathsf{Stack}}_{\mathsf{IFA}} \times \widehat{\mathsf{Implicit}} \times \mathsf{Dominators}$$

Elements of the analysis domain are written $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM)$ where the semicolon serves as a reminder that the dominator component, $DOM$, is a parameter to the Flow Logic specification and is not, as such, part of the analysis.

### 5.2.4 Flow Logic Specification

The information flow analysis is composed of three mostly independent components: a control flow analysis, tracking of implicit flows, and calculation of dominators. The control flow analysis could be divided into further two subcomponents: the pure control flow analysis, identical to previous control flow analyses, and a security label analysis that tracks the security levels of data manipulated by the program. For succinctness the three components are combined into a single specification with judgements of the form:

$$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{instr}$$

The full specification is shown in Figures 5.9, 5.10, and 5.11.

To enhance the readability of the Flow Logic specification several notational conveniences are used: the pattern match notation for abstract stacks is adapted for computing the current security level and bind it to a temporary variable, e.g., $\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1$. For local heaps the notation $\hat{L}_{\downarrow 1}(m, pc)(x)$ and $\hat{L}_{\downarrow 2}(m, pc)(x)$ is used to indicate the first component (abstract value) and second component (security level) of $\hat{L}(m, pc)(x)$ respectively. A similar notation is adopted for operand stacks and heaps.

When instantiating a new object the fields of the object must be filled with the default values and default security labels. The default security label for a field must be greater than or equal to the level given by the security policy, i.e., $f.level$. but it must also be greater than or equal to the level of the security context in which it is created otherwise the creation of an object could be used as a covert channel. Therefore the *default* function is modified to take a security label as an additional parameter:

$$\begin{array}{l} \forall f \in \sigma.fields\text{: } \exists \ell'\text{:} \\ \quad \ell \sqcup f.level \sqsubseteq \ell' \in \mathsf{Security} \wedge \\ \quad default_\ell(\sigma)(f) = \beta_{\mathsf{Const}}(def(f.type))^{\ell'} \end{array}$$

Below the Flow Logic specification (and calculation of post-dominators) is discussed in details for a few instructions.

#### Local Variables

Analysing the `load`-instruction first requires a calculation of the current security context:

$$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \hat{L}_{\downarrow 2}(m, pc)(x) :$$

Next the value of the local variable, $\hat{L}_{\downarrow 1}(m, pc)(x)$, is pushed on top of the stack, annotated with the new security label $\ell$:

$$\hat{L}_{\downarrow 1}(m, pc)(x)^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$$

The local heap was not modified and is simply copied forward

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$

the same holds for the implicit flows:

$$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$$

The `load`-instruction only has one post-dominator (in addition to itself), namely the immediately following instruction:

$$DOM(m, pc) = \{pc\} \cup DOM(m, pc + 1)$$

**Conditionals**

The information flow analysis for conditionals is very similar to the control flow analysis. First the two top values on the stack and their security labels are matched:

$$A_1{}^{\ell_1} :: A_2{}^{\ell_2} :: X \lhd \hat{S}(m, pc)$$

Based on the security levels of the stack values and the implicit flows the security level for the current instruction is calculated:

$$\ell \lhd \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2$$

Next the rest of the stack is pushed forward to the two possible jump destinations:

$$X \sqsubseteq \hat{S}(m, pc + 1)$$
$$X \sqsubseteq \hat{S}(m, pc_0)$$

and similarly for the local heap:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$
$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$$

Since conditionals give rise to new implicit flows that must be tracked, the current conditional is added to the set of tracked conditionals (and method invocations), all of which must also be copied forward:

$$\{(\ell, pc)\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$$
$$\{(\ell, pc)\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc_0)$$

The `if`-instruction is the only instruction that has non-trivial post-dominators. Since control flow may branch at a conditional the post-dominators for a conditional must be calculated as the intersection of the possible successor instructions (in addition to the node itself):

$$DOM(m, pc) = \{pc\} \cup (DOM(m, pc + 1) \cap DOM(m, pc_0))$$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{push}\ t\ n$
iff $\ell \triangleleft \sqcup \hat{C}(m, pc) :$
$\{\mathsf{INT}\}^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{pop}$
iff $A^{\ell} :: X \triangleleft \hat{S}(m, pc) :$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{load}\ t\ x$
iff $\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \hat{L}_{\downarrow 2}(m, pc)(x) :$
$\hat{L}_{\downarrow 1}(m, pc)(x)^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{store}\ x$
iff $A^{\ell_1} :: X \triangleleft \hat{S}(m, pc) :$
$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 :$
$A^{\ell} \sqsubseteq \hat{L}(m, pc+1)(x)$
$X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq_{\{x\}} \hat{L}(m, pc+1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{numop}\ t\ \mathit{binop}$
iff ${A_1}^{\ell_1} :: {A_2}^{\ell_2} :: X \triangleleft \hat{S}(m, pc) :$
$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2 :$
$\{\mathsf{INT}\}^{\ell} :: X \sqsubseteq \hat{S}(m, pc+1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)$

Figure 5.9: Information Flow Analysis (1)

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{if}\ t\ cmp\ \texttt{goto}\ pc_0$
iff $A_1{}^{\ell_1} :: A_2{}^{\ell_2} :: X \triangleleft \hat{S}(m, pc) :$
$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2 :$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$X \sqsubseteq \hat{S}(m, pc_0)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$
$\{(\ell, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$
$\{(\ell, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc_0)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{new}\ \sigma$
iff $\ell \triangleleft \sqcup \hat{C}(m, pc) :$
$\{(\mathsf{Ref}\ \sigma)\}^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$
$default_{\ell}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{getfield}\ f$
iff $B^{\ell_1} :: X \triangleleft \hat{S}(m, pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B :$
$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) :$
$\hat{H}_{\downarrow 1}(\mathsf{Ref}\ \sigma)(f)^{\ell} :: X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{putfield}\ f$
iff $A^{\ell_1} :: B^{\ell_2} :: X \triangleleft \hat{S}(m, pc) :$
$\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2 :$
$\forall(\mathsf{Ref}\ \sigma) \in B : A^{\ell} \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)(f)$
$X \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$

Figure 5.10: Information Flow Analysis (2)

$$
\begin{array}{l}
(\hat{H},\hat{L},\hat{S},\hat{C};DOM) \models_{\text{IFA}} (m,pc) : \texttt{invokevirtual}\ m_0 \\
\quad \text{iff} \quad A_1{}^{\ell_1} :: \cdots :: A_{|m_0|}{}^{\ell_{|m_0|}} :: B^{\ell_0} :: X \triangleleft \hat{S}(m,pc) : \\
\qquad \ell \triangleleft \sqcup \hat{C}(m,pc) : \\
\qquad \forall(\mathsf{Ref}\ \sigma) \in B : \\
\qquad\quad m_v \triangleleft \mathit{methodLookup}(m_0,\sigma) : \\
\qquad\quad \{(\mathsf{Ref}\ \sigma)\}^{\ell_0 \sqcup \ell} :: A_1{}^{\ell_1 \sqcup \ell} :: \cdots :: A_{|m_0|}{}^{\ell_{|m_0|} \sqcup \ell} \sqsubseteq \hat{L}(m_v,0)[0..|m_0|] \\
\qquad\quad \{(\ell_0,(m,pc))\} \cup \hat{C}(m,pc) \sqsubseteq_{DOM} \hat{C}(m_v,0) \\
\qquad\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad A^{\ell_A} :: Y \triangleleft \hat{S}(m_v,\mathsf{END}) : \\
\qquad\qquad\quad A^{\ell_A \sqcup \ell} :: X \sqsubseteq \hat{S}(m,pc+1) \\
\qquad\quad \hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\qquad\quad \hat{C}(m,pc) \sqsubseteq_{DOM} \hat{C}(m,pc+1) \\
\\
(\hat{H},\hat{L},\hat{S},\hat{C};DOM) \models_{\text{IFA}} (m,pc) : \texttt{return} \\
\quad \text{iff} \quad A^{\ell} :: X \triangleleft \hat{S}(m,pc) : \\
\qquad A^{\ell} :: X \sqsubseteq \hat{S}(m,pc)(m,\mathsf{END})
\end{array}
$$

Figure 5.11: Information Flow Analysis (3)

**Instance Fields**

For the `getfield` instruction a reference to the target object must first be obtained from the top of the stack:

$$B^{\ell_1} :: X \triangleleft \hat{S}(m,pc) :$$

For each of the (abstract) object references found the pertinent security level is computed and the relevant heap value is pushed on top of the stack:

$$
\begin{array}{l}
\forall(\mathsf{Ref}\ \sigma) \in B : \\
\quad \ell \triangleleft \sqcup \hat{C}(m,pc) \sqcup \ell_1 \sqcup \hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) : \\
\quad \hat{H}_{\downarrow 1}(\mathsf{Ref}\ \sigma)(f)^{\ell} :: X \sqsubseteq \hat{S}(m,pc+1)
\end{array}
$$

Note that the computed security level also takes the security level of the used object reference into account in order to prevent information about the object reference from leaking; thus preventing an implicit flow from the object reference.

Finally the local variables and implicit flows are computed in the usual way:

$$
\begin{array}{l}
\hat{L}(m,pc) \sqsubseteq \hat{L}(m,pc+1) \\
\hat{C}(m,pc) \sqsubseteq_{DOM} \hat{C}(m,pc+1)
\end{array}
$$

as are the post-dominators for the instruction:

$$DOM(m, pc) = \{pc\} \cup DOM(m, pc + 1)$$

**Method Invocation**

The information flow analysis for method invocation proceeds like the control flow analysis by matching stack parameters:

$$A_1{}^{\ell_1} :: \cdots :: A_{|m_0|}{}^{\ell_{|m_0|}} :: B^{\ell_0} :: X \triangleleft \hat{S}(m, pc) :$$

and calculating the security level of the context:

$$\ell \triangleleft \sqcup \hat{C}(m, pc) :$$

As for the control flow analysis all object references found on the stack are used for method lookup:

$$\forall(\mathsf{Ref}\ \sigma) \in B : \quad m_v = methodLookup(m_0, \sigma) \ldots$$

Next the parameters are transferred annotated with the updated security context:

$$\{(\mathsf{Ref}\ \sigma)\}^{\ell_0 \sqcup \ell} :: A_1{}^{\ell_1 \sqcup \ell} :: \cdots :: A_{|m_0|}{}^{\ell_{|m_0|} \sqcup \ell} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$$

and the implicit flows are also copied to the invoked method:

$$\{(\ell_0, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m_v, 0)$$

Since the specific method invoked depends on the object reference used, method invocation can be used for starting an implicit flow. Note however that such implicit flows only lasts for the duration of the method invocation since no matter which method was invoked they must all continue at the same instruction when they return, namely the instruction immediately following the `invokevirtual`-instruction. For the same reason it is not necessary to copy any implicit flows back from the invoked method since there is only one return statement in each method which is then trivially a post-dominator for all instructions in the method.

Any return values from the method invocation are handled as in the control flow analysis updated with the security level of the current context:

$$A^{\ell_A} :: Y \triangleleft \hat{S}(m_v, \mathsf{END}) : \quad A^{\ell_A \sqcup \ell} :: X \sqsubseteq \hat{S}(m, pc + 1)$$

Then the local heaps and (local) implicit flows are copied forward as usual:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$
$$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$$

Finally the dominator component is calculated:

$$DOM(m, pc) = \{pc\} \cup DOM(m, pc + 1)$$

Note that dominators are strictly local to a method and thus the dominators of a method invocation does *not* include any instructions in the invoked method. In security terms this means that methods invoked in a high security context can only modify fields, stack positions, and local variables classified as high security.

### 5.2.5 Soundness and Non-Interference

Proving that the information flow analysis is semantically sound amounts to showing that it can be used to show that a program is non-interfering in the sense of Definition 5.12. The previous analyses have been *first-order* analyses, cf [NNH99], where abstract values (properties) are used to directly describe sets of concrete values. In contrast, the information flow analysis is a *second-order* analysis where the abstract properties represent relations between values. This is a consequence of non-interference being a property on the set of all program executions rather than a property of individual program executions. Correspondingly the proof of correctness for the analysis is quite different and technically rather more involved than the proofs for previous analyses. For convenience the proof is staged through a number of technical definitions and lemmas that are discussed and proved in some detail in this section. In order to prove the main non-interference theorem (Theorem 5.24) it must first be established that the control flow and dominator parts of the analysis are correct. However, as already noted, the control flow part of the analysis is identical to the control flow analysis in Section 3.2 and thus the correctness was proved by Theorem 3.14. This leaves the dominator part which is proved correct by the following lemma:

**Lemma 5.16.** *Let* $P \in \mathsf{Program}$ *such that* $DOM(P)$ *and such that*

$$P \vdash C_1 \Longrightarrow C_2 \Longrightarrow \cdots \Longrightarrow C_n \Longrightarrow \langle H, \langle \mathsf{Ret}\ v \rangle \rangle$$

*where* $C_1 = \langle H_1, \langle m_1, pc_1, L_1, S_1 \rangle :: SF_1 \rangle$ *then*

$$\forall pc_1' \in DOM(m_1, pc_1)\colon \exists i \in \{1, \ldots, n\}\colon C_i = \langle H_i, \langle m_i, pc_1', L_i, S_i \rangle :: SF_i \rangle$$

*Proof.* By induction in $n$, the length of the reduction sequence. For the base step $n = 0$ the lemma holds vacuously. Now assume for the induction hypothesis that the lemma holds for $n = k$ and prove that it also holds for $n = k+1$. Thus assume that

$$P \vdash C_1 \Longrightarrow \cdots \Longrightarrow C_k \Longrightarrow C_{k+1} \Longrightarrow \langle H, \langle \mathsf{Ret}\ v \rangle \rangle$$

and apply the induction hypothesis to the sub-sequence

$$P \vdash C_2 \Longrightarrow \cdots \Longrightarrow C_k \Longrightarrow C_{k+1} \Longrightarrow \langle H, \langle \mathsf{Ret}\ v \rangle \rangle$$

which implies that

$$\begin{aligned} &\forall pc_2' \in DOM(m_2, pc_2)\colon \\ &\quad \exists i \in \{2, \ldots, k+1\}\colon C_i = \langle H_i, \langle m_i, pc_2', L_i, S_i \rangle :: SF_i \rangle \end{aligned} \tag{5.11}$$

The result now follows from a case analysis on the instruction performed in $C_1$:

*Case* `load`*:* It follows trivially from the semantics that $m_2 = m_1$. Now consider $pc'_1 \in DOM(m_1, pc_1)$. From $DOM(P)$ it follows that $DOM(m_1, pc_1) = \{pc_1\} \cup DOM(m_1, pc_2)$. If $pc'_1 = pc_1$ the lemma holds trivially. Assume that $pc'_1 \neq pc_1$ then it holds that $pc'_1 \in DOM(m_1, pc_1) \setminus \{pc_1\} \subseteq DOM(m_1, pc_2)$ and thus the case follows from equation (5.11).

*Case* `if`*:* It follows trivially from the semantics that $m_1 = m_2$. Let $pc'_1 \in DOM(m_1, pc_1)$. By inspection of the semantics it follows that $pc_2 = pc_1 + 1$ or $pc_2 = pc_0$ (for some $pc_0$) and from $DOM(P)$ it follows that $DOM(m_1, pc_1) = \{pc_1\} \cup (DOM(m_1, pc_1 + 1) \cap DOM(m_1, pc_0))$. If $pc'_1 = pc_1$ the lemma holds trivially. If $pc'_1 \neq pc_1$ then

$$\begin{aligned} pc'_1 &\in (DOM(m_1, pc_1 + 1) \cap DOM(m_1, pc_0)) \setminus \{pc_1\} \\ &\subseteq (DOM(m_1, pc_1 + 1) \cap DOM(m_1, pc_0)) \end{aligned}$$

From equation (5.11) it follows that

$$\begin{aligned} &\forall pc' \in DOM(m_1, pc_1 + 1): \\ &\quad \exists i \in \{2, \ldots, k+1\} : C_i = \langle H_i, \langle m_i, pc', L_i, S_i \rangle :: SF_i \rangle \end{aligned}$$

and

$$\begin{aligned} &\forall pc' \in DOM(m_1, pc_0): \\ &\quad \exists i \in \{2, \ldots, k+1\} : C_i = \langle H_i, \langle m_i, pc', L_i, S_i \rangle :: SF_i \rangle \end{aligned}$$

and thus

$$\begin{aligned} &\forall pc' \in DOM(pc_1 + 1) \cap DOM(pc_0): \\ &\quad \exists i \in \{2, \ldots, k+1\} : C_i = \langle H_i, \langle m_i, pc', L_i, S_i \rangle :: SF_i \rangle \end{aligned}$$

and therefore the case holds.

*Case* `invokevirtual`*:* It follows from the semantics that $S_1 = v_1 :: \cdots :: v_n :: loc :: S'_1$ for some $n$. Since all programs considered here are terminating the method invocation must return at some point and thus there must exist an $i$ such that $C_i = \langle H_i, \langle m_1, pc_1 + 1, L_1, S''_1 \rangle :: SF_1 \rangle$ where $S''_1 = v :: S'_1$ or $S''_1 = S'_1$ depending on whether or not the invoked returns a value or not. It now holds that $DOM(m_1, pc_1) = \{pc_1\} \cup DOM(m_1, pc_1 + 1)$ and the case proceeds in the same way as for the `load`-instruction.

The remaining cases are analogous and thus the lemma holds. ∎

The above lemma shows that the equations in Definition 5.15 do indeed specify the post-dominators for all addresses in a given program.

The proofs carried out in the preceding chapters rely on being able to define representation functions and correctness relations establishing that the analysis correctly (over-)approximates the data in a particular semantic configuration. For the information flow analysis the correctness relations must instead establish that the analysis correctly approximates the relation between data in two (or more) configurations and, by extension, program runs. For non-interference this

amounts to showing that all the "low data" of the two configurations is equivalent. Defining equivalence is complicated by the dynamic allocation of memory on the heap and therefore the equivalence is defined only up to isomorphism on the memory locations containing data of low security. Since locations are also data values that may occur in a program this isomorphism must also be taken into account when comparing values. This is formalised in the definition of the $\equiv_\pi$-equivalence, cf. Definition 5.10. With the above in mind the equivalence of two stack frames, modulo $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$, is defined as follows:

**Definition 5.17.** *Let $F_i = \langle m_i, pc_0, L_i, S_i \rangle$ for $i = 1, 2$ be stack frames and let $\mathcal{A} = (\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ then $F_1$ and $F_2$ are $\mathcal{A}$-equivalent, written $F_1 \approx^\pi_{\mathcal{A}} F_2$, if and only if $\pi : \mathsf{Loc} \to \mathsf{Loc}$ is a bijective partial map and the following conditions hold:*

1. $m_1 = m_2$
2. $pc_1 = pc_2$
3. $\forall x : \hat{L}_{\downarrow 2}(m_1, pc_1)(x) \sqsubseteq \mathsf{L} \Rightarrow L_1(x) \equiv_\pi L_2(x)$
4. $\forall i : \hat{S}_{\downarrow 2}(m_1, pc_1)|_i \sqsubseteq \mathsf{L} \Rightarrow S_1|_i \equiv_\pi S_2|_i$

This is trivially extended to call stacks: $SF_1 \approx^\pi_{\mathcal{A}} SF_2$ if and only if $\forall i : SF_1|_i \approx^\pi_{\mathcal{A}} SF_2|_i$. Note that this requires the two call stacks to be of equal length. Now the equivalence of two semantic configurations, modulo $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$, can be defined:

**Definition 5.18 ($\mathcal{A}$-equivalence).** *Let $C_i = \langle H_i, SF_i \rangle$ for $i = 1, 2$ be semantic configurations and let $\mathcal{A} = (\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ then $C_1$ and $C_2$ are* $\mathcal{A}$-equivalent*, written $C_1 \approx_{\mathcal{A}} C_2$, if and only if there exists an bijective partial map, $\pi : \mathsf{Loc} \to \mathsf{Loc}$, such that $SF_1 \approx^\pi_{\mathcal{A}} SF_2$ and for all $(\mathsf{Ref}\ \sigma) \in \mathrm{dom}(\hat{H})$ and $f \in \sigma.fields$ the following holds:*

$$
\begin{array}{l}
\hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) \sqsubseteq \mathsf{L} \Rightarrow \\
\quad \forall loc_1\colon\ H_1(loc_1).class = H_2(\pi(loc_1)).class \\
\qquad\qquad\ H_1(loc_1).f \equiv_\pi H_2(\pi(loc_1)).f \\
\quad \forall loc_2\colon\ H_2(loc_2).class = H_1(\pi^{-1}(loc_2)).class \\
\qquad\qquad\ H_2(loc_2).f \equiv_\pi H_1(\pi^{-1}(loc_1)).f
\end{array}
$$

Note that this definition is very similar to that for heap equivalence, cf. Definition 5.11, with added requirements on the local heap and the operand stack.

The pushing and popping on the operand stack introduces a minor technical problem since changes in the stack height also changes the (top-)indexing of stack positions leading to complex formulations of properties and proofs. To avoid this, special notation is introduced to index a stack from the bottom instead of from the top: let $\hat{S} = A_0 :: \cdots :: A_n$ and define $\hat{S}|^i = \hat{S}|_{n-i} = A_{n-i}$ for $i \in \{0, \ldots, n\}$. The lemma below shows that the two formulations are interchangeable for stacks of equal length:

**Lemma 5.19.** *If* $|\hat{S}(m, pc_1)| = |\hat{S}(m, pc_2)|$ *and*

$$\forall i : \hat{S}_{\downarrow 2}(m, pc_2)|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m, pc_1)|^i \sqsubseteq \mathsf{L}$$

*then*

$$\forall i : \hat{S}_{\downarrow 2}(m, pc_2)|_i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m, pc_1)|_i \sqsubseteq \mathsf{L}$$

*Proof.* Trivial. ∎

The next two lemmas show that variables and stack positions that are modified in a high security context will be marked as high-security in the analysis:

**Lemma 5.20.** *Let* $P \in \mathsf{Program}$ *and* $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{IFA} P$ *such that* $P \vdash C_1 \Longrightarrow C_2$ *with* $C_i.address = (m, pc_i)$ *for* $i = 1, 2$ *and* $m.instructionAt(pc_1) \neq$ `invokevirtual`. *If* $\sqcup \hat{C}(m, pc_1) = \mathsf{H}$ *then*

$$\forall i : \hat{S}_{\downarrow 2}(m, pc_2)|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m, pc_1)|^i \sqsubseteq \mathsf{L}$$

*and*

$$\forall x : \hat{L}_{\downarrow 2}(m, pc_2)(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m, pc_1)(x) \sqsubseteq \mathsf{L}$$

*Proof.* By case analysis.

*Case* `push`*:* By definition of the analysis it follows that $\hat{L}(m, pc_1) \sqsubseteq \hat{L}(m, pc_2)$ and therefore in particular:

$$\forall x : \hat{L}_{\downarrow 2}(m, pc_1)(x) \sqsubseteq \hat{L}_{\downarrow 2}(m, pc_2)(x)$$

which implies that

$$\forall x : \hat{L}_{\downarrow 2}(m, pc_2)(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m, pc_1)(x) \sqsubseteq \mathsf{L}$$

Now write $\hat{S}(m, pc_1) = A_0 :: \cdots A_m$ and $\hat{S}(m, pc_2) = A'_0 :: \cdots :: A'_n$ where $n = m + 1$ then it follows from the definition of the analysis that $\{\mathsf{INT}\}^\ell :: \hat{S}(m, pc_1) \sqsubseteq \hat{S}(m, pc_2)$ where $\ell = \mathsf{H}$ because $\sqcup \hat{C}(m, pc_1) = \mathsf{H}$ and thus

$$\{\mathsf{INT}\}^\ell :: A_0 :: \cdots :: A_m \sqsubseteq A'_0 :: \cdots :: A'_n$$

which leads to the conclusion that

$$\forall i : \hat{S}_{\downarrow 2}(m, pc_2)|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m, pc_1)|^i \sqsubseteq \mathsf{L}$$

The remaining cases are similar and thus the lemma holds. ∎

**Lemma 5.21.** *Let* $P \in \mathsf{Program}$ *and* $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{IFA} P$ *such that*

$$P \vdash C_1 \Longrightarrow \cdots \Longrightarrow C_n \Longrightarrow C_1' \Longrightarrow^* \langle H_1'', \langle \mathsf{Ret}\ v_1'' \rangle \rangle$$

*with* $C_j.address = (m_j, pc_j)$ *and* $C_1'.address = (m_1, pc_1')$. *If* $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_1, pc_1)$, $(m_1, pc_1'') \curvearrowright_i pc_1'$, *and* $m_j = m_1 \Rightarrow (m_1, pc_1'') \not\curvearrowright pc_j$ *for* $j \in \{1, \ldots, n\}$ *then for* $j \in \{1, \ldots, n\}$*:*

1. $\sqcup \hat{C}(m_j, pc_j) = \mathsf{H}$
2. $\hat{L}_{\downarrow 2}(m_1, pc_1')(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m_1, pc_1)(x) \sqsubseteq \mathsf{L}$
3. $\hat{S}_{\downarrow 2}(m_1, pc_1')|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m_1, pc_1)|^i \sqsubseteq \mathsf{L}$

*Proof.* By induction in the length of the derivation sequence, $n$.

*Base case* $n = 0$*:* For $n = 0$ the lemma reduces to $P \vdash C_1 \Longrightarrow C_1'$, such that $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_1, pc_1)$, $(m_1, pc_1'') \curvearrowright_i pc_1'$ with $(m_1, pc_1'') \not\curvearrowright pc_1$ and thus $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_1, pc_1) \Rightarrow \sqcup \hat{C}(m_1, pc_1)$. Since it trivially holds that $m_1.instructionAt(pc_1)$ can not be a method invocation, the case follows from Lemma 5.20.

*Induction Step* $n = k + 1$*:* For $n = k + 1$ it is assumed that $P \vdash C_1 \Longrightarrow \cdots \Longrightarrow C_{k+1} \Longrightarrow C_1'$, $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_1, pc_1)$, $(m_1, pc_1'') \curvearrowright_i pc_1'$, $m_j = m_1 \Rightarrow (m_1, pc_1'') \not\curvearrowright pc_j$ for $j \in \{1, \ldots, k+1\}$. There are two sub-cases:

*Sub-case* $m_1.instructionAt(pc_1) \neq$ `invokevirtual`*:* It follows that $m_1 = m_2$ and thus that $(m_1, pc_1'') \not\curvearrowright pc_2$ and therefore $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_1, pc_2)$. Thus the induction hypothesis can be applied to $P \vdash C_2 \Longrightarrow \cdots \Longrightarrow C_{k+1} \Longrightarrow C_1'$ which results in $\sqcup \hat{C}(m_j, pc_j) = \mathsf{H}$ for $j \in \{2, \ldots, k+1\}$ (and by assumption also $\sqcup \hat{C}(m_1, pc_1) = \mathsf{H}$) and

$$\forall x : \hat{L}_{\downarrow 2}(m_1, pc_1')(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m_1, pc_2)(x) \sqsubseteq \mathsf{L}$$

and

$$\forall i : \hat{S}_{\downarrow 2}(m_1, pc_1')|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m_1, pc_2)|^i \sqsubseteq \mathsf{L}$$

The case now follows from Lemma 5.20.

*Sub-case* $m_1.instructionAt(pc_1) =$ `invokevirtual`*:* It follows from the semantics that $m_1 \neq m_2$ and from the definition of the analysis that $(\mathsf{H}, (m_1, pc_1'')) \in \hat{C}(m_2, 0)$. Since recursive method invocations are not allowed (see Remark 5.8) and since the execution sequence terminates by assumption the invoked method must return at some point and thus there must exist an $i \in \{3, \ldots, n\}$ such that $C_i.address = (m_2, pc_2')$ containing a `return` instruction: $m_2.instructionAt(pc_2') =$ `return` $t$ and such that $C_{i+1}.address = (m_1, pc_1 + 1)$. The case now follows from applying the induction hypothesis to $P \vdash C_{i+1} \Longrightarrow^* C_1'$ and thus obtain

$$\forall x : \hat{L}_{\downarrow 2}(m_1, pc_1')(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m_1, pc_1 + 1)(x) \sqsubseteq \mathsf{L}$$

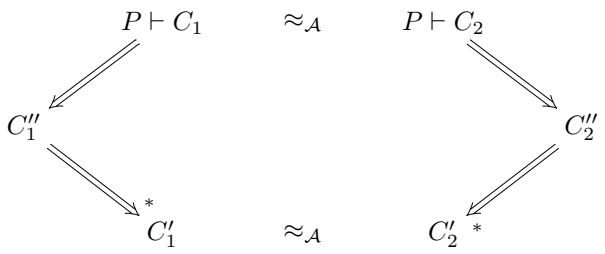


Figure 5.12: Diamond property

and

$$\forall i : \hat{S}_{\downarrow 2}(m_1, pc_1')|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m_1, pc_1 + 1)|^i \sqsubseteq \mathsf{L}$$

From the definition of the analysis it follows that

$$\forall x : \hat{L}_{\downarrow 2}(m_1, pc_1 + 1)(x) \sqsubseteq \mathsf{L} \Rightarrow \hat{L}_{\downarrow 2}(m_1, pc_1)(x) \sqsubseteq \mathsf{L}$$

and

$$\forall i : \hat{S}_{\downarrow 2}(m_1, pc_1 + 1)|^i \sqsubseteq \mathsf{L} \Rightarrow \hat{S}_{\downarrow 2}(m_1, pc_1)|^i \sqsubseteq \mathsf{L}$$

Note that the invoked method will execute all its instructions in a high-context since, by definition of '$\sqsubseteq_{DOM}$', implicit flow can only be removed in the method where they originate. Thus all local variables, stack positions, and heap references modified by the invoked method will all be marked as high-security and thus have no influence on the low-equivalence.

This concludes the proof. ∎

It is now possible to state and prove the main technical lemma of this section. The lemma (called a "hexagon lemma" in [CKP03]) shows that $\mathcal{A}$-equivalence on configurations is preserved by reduction or, more precisely, that $\mathcal{A}$-equivalence is preserved by sufficiently long reduction sequences. Figure 5.12 summarises the lemma and illustrates the source of its name. Note that this proof works on the assumption that programs are bytecode verified and thus that the stack height is fixed for each instruction.

**Lemma 5.22 (Diamond property).** *Let* $P \in \mathsf{Program}$, $\mathcal{A} \in \widehat{\mathsf{Analysis}_{\mathsf{IFA}}}$, $C_1, C_2 \in \mathsf{Conf}$ *such that* $\mathcal{A} \models_{IFA} P$ *and* $P \vdash C_1 \Longrightarrow C_1'' \Longrightarrow^* \langle H_1''', \langle \mathsf{Ret}\ v_1''' \rangle \rangle$, $P \vdash C_2 \Longrightarrow C_2'' \Longrightarrow^* \langle H_2''', \langle \mathsf{Ret}\ v_2''' \rangle \rangle$ *with* $C_1 \approx_{\mathcal{A}} C_2$ *then* $\exists C_1', C_2'$ *such that* $P \vdash C_1'' \Longrightarrow^* C_1'$, $P \vdash C_2'' \Longrightarrow^* C_2'$, *and* $C_1' \approx_{\mathcal{A}} C_2'$.

*Proof.* By case analysis.

*Case* `load`*:* By assumption $C_i = \langle H_i, \langle m, pc, L_i, S_i \rangle :: SF_i \rangle$ and for a `load` instruction this results in $C_i'' = \langle H_i, \langle m, pc + 1, L_i, L_i(x) :: S_i \rangle :: SF_i \rangle$ for $i = 1, 2$. There are two sub-cases:

*Sub-case* $\hat{S}_{\downarrow 2}(m, pc+1) = \mathsf{H}$*:* It is trivially true that

$$\forall i : \hat{S}_{\downarrow 2}(m, pc+1)|_i \sqsubseteq \mathsf{L} \Rightarrow (L_1(x) :: S_1)|_i \equiv_\pi (L_2(x) :: S_2)|_i$$

since by assumption $\forall i : \hat{S}_{\downarrow 2}(m, pc)|_i \sqsubseteq \mathsf{L} \Rightarrow S_1|_i \equiv_\pi S_2|_i$.

*Sub-case* $\hat{S}_{\downarrow 2}(m, pc_1 + 1) \sqsubseteq \mathsf{L}$*:* By definition of the information flow analysis it follows that

$$\sqcup \hat{C}(m, pc) \sqcup \hat{L}_{\downarrow 2}(m, pc)(x) \sqsubseteq \hat{S}_{\downarrow 2}(m, pc+1)|_0$$

and thus $\hat{L}_{\downarrow 2}(m, pc)(x) \sqsubseteq \mathsf{L}$ but by assumption it then follows that $L_1(x) \equiv_\pi L_2(x)$ and thus

$$\forall i : \hat{S}_{\downarrow 2}(m, pc+1)|_i \sqsubseteq \mathsf{L} \Rightarrow (L_1(x) :: S_1)|_i \equiv_\pi (L_2(x) :: S_2)|_i$$

Thus $C_1'' \approx_{\mathcal{A}} C_2''$. The lemma now holds by taking $C_1' = C_1''$ and $C_2' = C_2''$.

*Case* `new`*:* It follows from the assumptions that $C_i'' = \langle H_i[loc_i \mapsto o_i], \langle m, pc + 1, L_i, loc_i :: S_i\rangle :: SF_i\rangle$ such that $loc_i \notin \mathrm{dom}(H_i)$ for $i = 1, 2$. Let $\pi$ be the map existing by the assumptions and define $\pi' = \pi[loc_1 \mapsto loc_2]$ then $H_1(loc_1).class = \sigma = H_2(\pi(loc_1)).class$ and since $o_1$ and $o_2$ are newly instantiated objects of the same class:

$$\forall f : H_1(loc_1).f = H_2(loc_2).f$$

Thus $C_1'' \approx_{\mathcal{A}} C_2''$ and the lemma holds.

*Case* `if`*:* By assumption $C_i = \langle H_i, \langle m, pc, L_i, v_i :: w_i :: S_i\rangle :: SF_i\rangle$ and thus $C_i'' = \langle H_i, \langle m, pc_i'', L_i, S_i\rangle\rangle$ where either $pc_i'' = pc + 1$ or $pc_i'' = pc_0$ for $i = 1, 2$. There are two sub-cases: $pc_1'' = pc_2''$ or $pc_1'' \neq pc_2''$. The former is straightforward and only the latter is detailed here. Since $pc_1'' \neq pc_2''$ it must be the case that $v_1 \not\equiv_\pi v_2$ or $w_1 \not\equiv_\pi w_2$ and thus that

$$\hat{S}_{\downarrow 2}(m, pc)|_0 \sqcup \hat{S}_{\downarrow 2}(m, pc)|_1 = \mathsf{H}$$

and thus by definition of the analysis that $(\mathsf{H}, (m, pc)) \in \hat{C}(m, pc+1)$ and $(\mathsf{H}, (m, pc)) \in \hat{C}(m, pc_0)$. Then by choosing $C_1'$ and $C_2'$ such that $pc_1' = pc_2'$ (and thus the stack heights of $C_1'$ and $C_2'$ are equal as required by the bytecode verifier) and $(m, pc) \curvearrowright_i pc_1'$, i.e., by choosing the first confluence point of the two reduction sequences, it follows from lemmas 5.19 and 5.21 that $C_1' \approx_{\mathcal{A}} C_2'$. Note that since only terminating execution sequences are considered all instructions *must* have an immediate post-dominator.

*Case* `invokevirtual`*:* By assumption $C_i = \langle H_i, \langle m, pc, L_i, v_{i,1} :: \cdots :: v_{i,|m_0|} :: loc_i :: S_i\rangle :: SF_i\rangle$ and thus

$$C_i'' = \langle H_i, \langle m_i'', 0, [0 \mapsto loc_i, 1 \mapsto v_{i,1}, \ldots, |m_0| \mapsto v_{i,|m_0|}], \epsilon\rangle :: SF_i''\rangle$$

If $m_1'' = m_2''$ the case is straightforward and follows from the definition of the analysis. Therefore only the case $m_1'' \neq m_2''$ is detailed here. From $m_1'' \neq m_2''$ it follows that $loc_1 \not\equiv_\pi loc_2$ and thus $(\mathsf{H}, (m, pc)) \in \hat{C}(m_1'', 0)$ and $(\mathsf{H}, (m, pc)) \in \hat{C}(m_2'', 0)$. Now note that $(m, pc) \curvearrowright_i pc + 1$ and define $C_i' = \langle H_i', \langle m, pc + 1, L_i', S_i' \rangle :: SF_i \rangle$. Since only terminating execution sequences are considered it follows that

$$P \vdash C_1'' \Longrightarrow^* C_1' \qquad \text{and} \qquad P \vdash C_2'' \Longrightarrow^* C_2'$$

It then follows from lemmas 5.19 and 5.21 that $C_1' \approx_{\mathcal{A}} C_2'$.

The remaining cases are similar. ∎

Having established the diamond property for $\mathcal{A}$-equivalence all that remains is to relate the security policy for a program to the security levels found by the information flow analysis:

**Definition 5.23 (Security compatible).** *For a program, $P \in \mathsf{Program}$, such that $(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{CFA} P$ the analysis, $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM)$, is said to be* security compatible *with $P$ if*

$$\forall \sigma \in P.classes\text{: } \forall f \in \sigma.fields\text{: } \quad f.level = \mathsf{L} \Rightarrow \hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) = \mathsf{L}$$

Finally, the main non-interference result can be stated and proved:

**Theorem 5.24 (Non-Interference).** *Let $P \in \mathsf{Program}$ and $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ such that $\mathcal{A} \models_{CFA} P$ and $\mathcal{A}$ is security compatible with $P$. If $C_0$ and $C_0'$ are initial configurations for $P$ such that $C_0 \approx_{\mathcal{A}} C_0'$ and $P \vdash C_0 \Longrightarrow^* \langle H, \langle \mathsf{Ret}\ v \rangle \rangle$ and $P \vdash C_0' \Longrightarrow^* \langle H', \langle \mathsf{Ret}\ v' \rangle \rangle$ then $P$ is non-interfering, i.e., $H \approx_{\mathsf{L}} H'$.*

*Proof.* Follows directly by application of Lemma 5.22 and Definition 5.18. ∎

The theorem shows that if the security level found by the information flow analysis agrees with those of the given security policy for a given program, then the program is non-interfering and thus no secret information can be leaked.

## 5.3 Summary

The analyses developed in this chapter fall within the main themes of this dissertation: safety and security.

Section 5.1 presented a *transaction flow analysis* that computes an over-approximation of the possible transaction depths in which a given instruction is execute. The analysis is then used to verify that the use of transactions in a program is *well-formed* and does not lead to runtime errors. In order to be useful for such verification the analysis is made context dependent with transaction used as context. The transaction flow analysis was developed for Carmel Core. In [HS05] the analysis is extended to cover Carmel Core with exceptions. The well-formedness of transactions is also examined in [PBB+04] as an

example of using the Java Modelling Language (JML), cf. [BCC+, JML05], to specify different high-level security properties and automatically derive appropriate JML annotations. Verification of a property is then done using a theorem prover on the JML annotations. For simple annotations, like those for nested transactions, it is possible to obtain fully automated proofs. The JML approach works at the level of Java Card, not Java Card bytecode (JCVML). The Java Card transaction model in general is described in [Siv03a, Sun00] and [Oes99] discusses various shortcomings and implementation strategies for transaction on Java smart cards.

The *information flow analysis* presented in Section 5.2 tracks the possible flow of information to and from instance fields. In particular the analysis ensures that no instance field with a low security classification depends on any instance field with a high security classification, i.e., that the high security fields are *non-interfering* with the low security fields. By concentrating on information flow between fields, rather than between local variables or method parameters and return values, the notion of non-interference defined here differ from those used in other work, cf. [KS02, ABF03, GS05]. This leads to a notion of security that closer, at least conceptually, to that of [FSBJ97, SBCJ97]. The purpose being to obtain a more flexible and practical definition of security easier to apply to real-world problems and thereby meet some of the points raised in [RMMG01].

In [BCG+00] model checking is used to guarantee that no "illicit" applet interactions can take place and thereby leak secret information. A type-system for guaranteeing non-interference (and pointer-confinement) for a Java Card like subset of Java is specified and proved correct in [BN02]. Similarly a (non-standard) type-system is used in [BS99] to perform information flow analysis and guarantee non-interference for the $FOb_{1<:}$ object calculus. An overview of language-based information flow security is given in [SM03] and [Rus92] gives a thorough introduction to non-interference (both transitive and in-transitive).

Chapter 6

# The Devil's Advocate

*Better the devil you know than the devil you don't know*
*—English Proverb*

The analyses presented in the previous chapters all work under the proviso that the *whole program* to be analysed is known and available *a priori*. Such a requirement is becoming increasingly untenable as more and more systems allow or even encourage extensions to be added dynamically. This is especially true for the Java Card platform that expressly allows and supports dynamic download of applets onto a Java smart card even after it has been issued to an end user. The flexibility afforded by these systems enables the development of highly adaptable and versatile solutions where additional functionality is implemented by simply adding new applets. However, that flexibility also greatly impacts the design and development of analyses and tools for optimisation and verification since it has become impossible, in general, to know the specific context in which an applet may run. A analogous problem is encountered when analysing runtime libraries where the "client" of a library is unknown beforehand. Therefore analysis of systems that can be extended dynamically at runtime (and analysis of libraries) presents a fundamental problem for solutions based on whole-program analysis.

In this chapter a so-called *devil's advocate* (or *hardest attacker*), cf. [NN00, Han04], is presented for a simplified control flow analysis. A devil's advocate is an abstract representation of (the analysis result of) all possible dynamic extensions of the system under analysis (relative to a given analysis), i.e., the abstract behaviour of all possible extensions as seen from the perspective of a simplified control flow analysis. Consequently the devil's advocate can be used to represent the applets that are downloaded dynamically and thus it provides a solution to the above mentioned problem. In the following this approach is illustrated by applying it to the problem of *leaking references* discussed in Section 4.2: by analysing a system in conjunction with the concomitant devil's

advocate it can be guaranteed that certain object references are never leaked to any other applet. This approach is inspired by the work on firewall validation in the ambient calculus [NNH02, NNHJ99].

## 6.1 Carmel Core and Program Extensions

To facilitate the presentation this chapter is based on Carmel Core, rather than full Carmel. To further simplify the presentation packages are dispensed with and thus an applet is comprised directly by a set of classes. The notion of sharing defined in Section 4.2.3 is re-defined accordingly as are initial configurations:

**Definition 6.1 (Initial configurations).** *For a program, $P \in \mathsf{Program}$, the configuration $C \in \mathsf{RunConf}$ is an* initial configuration *if and only if $\sigma \in P.main$, $m_\sigma = \sigma.entry$, $\sigma.sharing = \sigma_1 :: \cdots :: \sigma_n$, $\sigma_i \in P.main$ for $1 \leq i \leq n$, $C = \langle H, \langle m_\sigma, 0, [0 \mapsto loc_\sigma, 1 \mapsto loc_{\sigma_1}, \ldots, n \mapsto loc_{\sigma_n}], \epsilon\rangle :: \epsilon\rangle$ and $\forall \tau \in P.main : \exists loc_\tau : H(loc_\tau) = \tau$.*

Note that since Carmel Core does not include instructions for manipulating the static heap it is not included in the semantic configurations, cf. Section 2.6.

**Remark 6.2 (Syntactic conventions).** *Figure 6.1 shows an example Carmel Core program that illustrates the syntactic conventions used in this chapter:*

1. *All classes are main classes.*
2. *The entry point of class $\sigma$ is called* `m_`$\sigma$*, e.g.,* `Bob`.*entry* = `m_Bob`.
3. *Parameters to entry point methods indicate sharing, e.g.,* `m_Alice(Bob)` *means* `Alice`.*sharing* = `Bob`.

**Example 6.3.** *In Figure 6.1 an example Carmel program, $P_{AB}$, is shown that illustrates a typical situation for Java Card applications: two applets (represented by the classes* `Alice` *and* `Bob` *respectively) wish to communicate in the form of a method invocation from* `Alice` *to* `Bob`*. The method invocation requires* `Alice` *to have access to an object reference to* `Bob` *which is obtained by* sharing*. As noted in Remark 6.2* `Alice` *uses a reference parameter to* `Bob` *in the entry method* `m_Alice` *to indicate that it wants to obtain a reference to* `Bob` *through sharing.*

*Note that by passing an* `Alice`*-reference on to* `Bob`*,* `Alice` *grants* `Bob` *access to* `Alice`*. Once the two applets are done communicating* `Alice` *wants to make sure that* `Bob` *does not leak the* `Alice`*-reference to any other applets.*

The analyses of the previous chapters were developed with the underlying assumption that the whole program is available *a priori* for analysis. As mentioned in the introduction to this chapter such as assumption is rather restrictive for a platform like JCVML where post-issuance dynamic download of applets is supported directly by the platform. In this chapter that restriction is lifted and a control flow analysis is developed to handle dynamic program extensions.

First the semantics must be extended to cover *program extensions*. For the present purpose a relatively simple definition will suffice:

```
class Alice {
  void m_Alice(Bob) {
    0: load r 1
    1: load r 0
    2: invokevirtual Bob.update(Object)
    3: return
  }
  /* ... */
}
```

```
class Bob {
  Object cache;

  void m_Bob() {
    0: return
  }
  void update(Object) {
    0: load r 0
    1: load r 1
    2: putfield Bob.cache
    /* ... */
    3: return
  }
}
```

Figure 6.1: Example program $P_{AB}$

```
class Mallet {
  void m_Mallet(Bob) {
    0: load r 1
    1: getfield Bob.cache
    2: return
  }
}
```

Figure 6.2: "Malicious" program $P_M$.

```
class Charlie {
  void m_Charlie(Bob) {
    0: load r 1
    1: invokevirtual Bob.m_Bob()
    2: return
  }
}
```

Figure 6.3: "Innocuous" program $P_C$.

**Definition 6.4 (Program extension).** *For* $P, Q \in \mathsf{Program}$ *such that* $\forall \sigma \in P.classes: \forall \tau \in Q.classes: \sigma \not\preceq \tau$, $P.sharing = \sigma_1 :: \cdots :: \sigma_m \Rightarrow \sigma_i \in P.main$ *for* $1 \leq i \leq m$, *and* $Q.sharing = \tau_1 :: \cdots :: \tau_n \Rightarrow \tau_j \in P.main \cup Q.main$ *for* $1 \leq j \leq n$, *the* extension *of* $P$ *with* $Q$, *written* $P\langle Q \rangle \in \mathsf{Program}$, *is defined as* $P\langle Q \rangle.classes = P.classes \cup Q.classes$.

A *program extension* is the union of the classes defined in the program and its extension with the added requirement that none of the main programs classes inherit from any of the classes in the extension, i.e., the class hierarchy of the main program can not be changed only extended. Note that the *methodLookup* does not need to be changed since it works "bottom up", i.e., from classes and methods up to packages and programs.

**Example 6.5.** *Figures 6.2 and 6.3 each show an applet used to extend the program* $P_{AB}$ *in Figure 6.1. The security properties of the extended programs* $P_{ABC} = P_{AB}\langle P_C \rangle$ *and* $P_{ABM} = P_{AB}\langle P_M \rangle$ *will be examined in the next section.*

## 6.2 Leaking References

Intuitively a class, $\tau$, is said to have been *leaked* to another class, $\sigma$, if there is an object of class $\sigma$ that contains either in an instance field or in the local heap or on the operand stack a reference to $\tau$. Formalising this intuition $v \in S$ is used as a shorthand for an operand stack $S \in \mathsf{Stack}$ if $S = v_1 :: \cdots :: v_n$ and $\exists i{:}\, v = v_i$. Similarly $v \in L$ is written for $L \in \mathsf{LocHeap}$ if $\exists x \in \mathrm{dom}(L){:}\, L(x) = v$. It can now formally be defined when a class has been leaked in a given semantic configuration:

**Definition 6.6 (Leaked).** *Given a configuration $\langle H, SF\rangle \in \mathsf{Conf}$ a class $\tau$ is said to be* leaked to class $\sigma$ *in $\langle H, SF\rangle$, written $\langle H, SF\rangle \vdash \tau \rightsquigarrow \sigma$, if and only if $\exists loc_\tau{:}\, \exists loc_\sigma{:}\, \exists f{:}\, H(loc_\tau).class = \tau \wedge H(loc_\sigma).class = \sigma \wedge H(loc_\sigma).f = loc_\tau$ or $\exists loc_\tau{:}\, \exists \langle m, pc, L, S\rangle \in SF{:}\, H(loc_\tau).class = \tau \wedge m.class = \sigma \wedge (loc_\tau \in S \vee loc_\tau \in L)$.*

This definition can then be extended to cover program executions:

**Definition 6.7 (Leaks).** *A program $P \in \mathsf{Program}$ is said to* leak *the class $\tau$ to class $\sigma$, written $P \vdash \tau \rightsquigarrow \sigma$, if and only if there exists configurations $C_0$ and $C$ such that $C_0$ is an initial configuration for $P$ and $P \vdash C_0 \Longrightarrow^* C$ with $C \vdash \tau \rightsquigarrow \sigma$.*

The notation $P \vdash \tau \not\rightsquigarrow \sigma$ denotes that program $P$ does *not* leak $\tau$ to $\sigma$.

**Example 6.8.** *By executing the program $P_{ABM}$, i.e., program $P_{AB}$ in Figure 6.1 extended with program $P_M$ from Figure 6.2, in such a way that both* `Alice` *and* `Mallet` *may communicate with* `Bob`*: $P_{ABM}.sharing($`Alice`$) =$ `Bob` *and $P_{ABM}.sharing($`Mallet`$) =$ `Bob`. Then by executing the program it is easy to see that $P_{ABM} \vdash$ `Alice` $\rightsquigarrow$ `Mallet`. The leak is made possible because* `Bob` *caches the* `Alice`*-reference in an instance field that is suddenly accessible to and read by the program extension* `Mallet`*.*

*Consider on the other hand program $P_C$ that does not read the* `cache` *field of* `Bob` *(even though is has access to do so) and therefore, intuitively $P_{ABC} \vdash$* `Alice` $\not\rightsquigarrow$ `Charlie`*. However, in order to prove that* `Alice` *is not leaked to* `Charlie` all *possible program executions must be tried and checked.*

It should be evident that since the control flow analysis defined in Section 3.2 specifies a conservative approximation of *all* possible executions of a program it can be used in an obvious way to check if a given program, e.g., $P_{ABC}$ in the above example, leaks a particular reference by inspecting the analysis results. This is discussed further in Section 6.3.3 below.

## 6.3 Control Flow Analysis

In this section a special control flow analysis is designed for Carmel Core. While it is somewhat simpler and less precise than the analysis defined in Section 3.2 it has the additional feature of being parameterised on two equivalence relations: one on classes, $\equiv_C \subseteq \mathsf{Class} \times \mathsf{Class}$, and the other on methods,

$\equiv_M \subseteq \mathsf{Method} \times \mathsf{Method}$. The corresponding characteristic maps takes a class or a method respectively to its equivalence class:

$$[\cdot]_C : \mathsf{Class} \to \mathsf{Class}_{/\equiv_C} \qquad [\cdot]_M : \mathsf{Method} \to \mathsf{Method}_{/\equiv_M}$$

In a later section these equivalence relations are used to partition the infinite sets of classes and methods into a finite number of equivalence classes thereby enabling us to define a finite *devil's advocate*. Moreover, the partitioning can be used to fine tune the precision of the analysis: by choosing fewer equivalence classes the less precise the analysis and vice versa.

### 6.3.1 Abstract Domains

The abstract domains are quite similar to those of previous analyses, however, modified to take the equivalence relations into account.

For abstract object references the equivalence relation over classes is incorporated by taking the set of equivalence classes to be the domain of abstract object references:

$$\overline{\mathsf{ObjRef}}_{\mathsf{DA}} = \mathsf{Class}_{/\equiv_C}$$

Abstract object references are written $[\mathsf{Ref}\ \sigma]_C$.

The domains for abstract values are changed to reflect the new definition of abstract object references:

$$\overline{\mathsf{Val}}_{\mathsf{DA}} = \overline{\mathsf{ObjRef}}_{\mathsf{DA}} + \{\mathsf{INT}\} \qquad \widehat{\mathsf{Val}}_{\mathsf{DA}} = \mathcal{P}(\overline{\mathsf{Val}}_{\mathsf{DA}})$$

A very simple representation is chosen for abstract objects:

$$\widehat{\mathsf{Object}}_{\mathsf{DA}} = \widehat{\mathsf{Val}}_{\mathsf{DA}}$$

In contrast to previous definitions the above does not distinguish between the different instance fields of an object and simply represents an object as the union of all its fields.

The abstract heap is updated accordingly:

$$\widehat{\mathsf{Heap}}_{\mathsf{DA}} = \widehat{\mathsf{ObjRef}}_{\mathsf{DA}} \to \widehat{\mathsf{Object}}_{\mathsf{DA}}$$

In contrast to previous analyses where an abstract local heap is assigned to each instruction in a method, this analysis only tracks local heaps for each of the equivalence classes over the domain of methods:

$$\widehat{\mathsf{LocHeap}}_{\mathsf{DA}} = \mathsf{Method}_{/\equiv_M} \to \widehat{\mathsf{Val}}_{\mathsf{DA}}$$

And similarly for operand stacks:

$$\widehat{\mathsf{Stack}}_{\mathsf{DA}} = \mathsf{Method}_{/\equiv_M} \to \widehat{\mathsf{Val}}_{\mathsf{DA}}$$

Combining all of the above results in the following domain for the *devil's advocate* analysis:

$$\widehat{\mathsf{Analysis}}_{\mathsf{DA}} = \widehat{\mathsf{Heap}}_{\mathsf{DA}} \times \widehat{\mathsf{LocHeap}}_{\mathsf{DA}} \times \widehat{\mathsf{Stack}}_{\mathsf{DA}}$$

In the next section the specification of a control flow analysis based on the above abstract domains is defined and discussed.

$$
\begin{array}{lll}
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{new}\ \sigma & \text{iff} & \{[\mathsf{Ref}\ \sigma]_C\} \subseteq \hat{S}([m]_M) \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{load}\ t\ x & \text{iff} & \hat{L}([m]_M) \subseteq \hat{S}([m]_M) \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{store}\ t\ x & \text{iff} & \hat{S}([m]_M) \subseteq \hat{L}([m]_M) \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{push}\ t\ n & \text{iff} & \{\mathsf{INT}\} \subseteq \hat{S}([m]_M) \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{pop}\ n & \text{iff} & \mathit{true} \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{numop}\ \mathit{op}\ t & \text{iff} & \mathit{true} \\
(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{if}\ t\ \mathit{cmp}\ \texttt{goto}\ pc_0 & \text{iff} & \mathit{true}
\end{array}
$$

$(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{getfield}\ f$
iff $\forall [\mathsf{Ref}\ \sigma]_C \in \hat{S}([m]_M)$: $\hat{H}([\mathsf{Ref}\ \sigma]_C) \subseteq \hat{S}([\mathsf{Ref}\ \sigma]_C)$

$(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{putfield}\ f$
iff $\forall [\mathsf{Ref}\ \sigma]_C \in \hat{S}([m]_M)$: $\hat{S}([\mathsf{Ref}\ \sigma]_C) \subseteq \hat{H}([\mathsf{Ref}\ \sigma]_C)$

$(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{return}\ t$ iff $\mathit{true}$

$(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{invokevirtual}\ m_0$
iff $\forall [\mathsf{Ref}\ \sigma]_C \in \hat{S}([m]_M)$:
$\forall [m_v]_M \in \mathit{methodLookup}_{/\equiv}(m_0, [\mathsf{Ref}\ \sigma]_C)$:
$\hat{S}([m]_M) \subseteq \hat{L}([m_v]_M)$
$[m_v]_M.\mathit{returnVal} \Rightarrow \hat{S}([m_v]_M) \subseteq \hat{S}([m]_M)$

Figure 6.4: Flow Logic specification for *devil's advocate* analysis

### 6.3.2 Flow Logic Specification

The judgements of the *devil's advocate* analysis are of the form:

$$(\hat{H},\hat{S},\hat{L}) \models_{\text{DA}} (m,pc) : \texttt{instr}$$

where $(\hat{H},\hat{S},\hat{L}) \in \widehat{\mathsf{Analysis}}_{\mathsf{DA}}$, $(m,pc) \in \mathsf{Addr}$, and $\texttt{instr} \in \mathsf{Instruction}$. The full Flow Logic specification is given in Figure 6.4 and a few of the clauses are examined in detail in the following.

The `new` instruction allocates space for an object in the heap and puts a reference to that space on top of the stack. In the analysis only the latter is modelled:

$$\{[\mathsf{Ref}\ \sigma]_C\} \subseteq \hat{S}([m]_M)$$

Note that since this analysis does not model stack positions and merges the abstract stack for all instructions in a given method there is no need to copy forward the stack as was the case for the previous analyses. Similarly for local heaps.

Modelling the `load` instruction in this analysis simply means copying the entire abstract local heap into the abstract stack:

$$\hat{L}([m]_M) \subseteq \hat{S}([m]_M)$$

Conversely for the `store` instruction:

$$\hat{S}([m]_M) \subseteq \hat{L}([m]_M)$$

Since `push` is only allowed to push numerical constants, and not the `null`-reference, the analysis of `push` is reduced to the following:

$$\{\mathsf{INT}\} \subseteq \hat{S}([m]_M)$$

The simple structure of the abstract stack, i.e., the lack of stack positions, means that the `pop` instruction has no discernible effect in the analysis. This is specified by defining the analysis of `pop` to be *true*, meaning that this instruction does not add any requirements on the analysis result. A similar argument holds for the `numop`, `if`, and `return` instructions.

The analyses designed earlier were able to leverage the *methodLookup* function, defined in the semantics, when analysing the `invokevirtual` instruction. However, the introduction of equivalence relations over methods and classes makes the direct use of *methodLookup* impossible. Instead an abstract model of dynamic dispatch must be modelled taking the equivalence relations into account:

$$\begin{aligned} &\mathit{methodLookup}_{/\equiv}(m_0, [\sigma]_C) = \\ &\quad \{[m_v]_M \mid m_v = \mathit{methodLookup}(m_0, \sigma'), \sigma' \in [\sigma]_C\} \end{aligned} \tag{6.1}$$

The $\mathit{methodLookup}_{\equiv}$ function is a trivial over-approximation of the semantic *methodLookup* function in the sense that $m_v = \mathit{methodLookup}(m_0, \sigma)$ implies $[m_v]_M \in \mathit{methodLookup}_{/\equiv}(m_0, [\sigma]_C)$.

This, however, introduces another problem concerning the return values of methods: it is no longer possible to directly check if a given method has a return type different from `void`. Again an over-approximation is defined for the underlying semantic functions:

$$[m]_M.\mathit{returnVal} = \begin{cases} \mathit{true} & \text{if } \exists m' \in [m]_M : m'.\mathit{returnType} \neq \texttt{void} \\ \mathit{false} & \text{otherwise} \end{cases} \tag{6.2}$$

With the above auxiliary functions in place the analysis of `invokevirtual` is straightforward. First all object references on the stack are located

$$\forall [\mathsf{Ref}\ \sigma]_C \in \hat{S}([m]_M) : \ldots$$

these are then used to look up the actual method to be invoked:

$$\forall [m_v]_M \in \mathit{methodLookup}_{/\equiv}(m_0, [\mathsf{Ref}\ \sigma]_C) : \ldots$$

Once a method is located the arguments are transferred to the local heap of the invoked method:

$$\hat{S}([m]_M) \subseteq \hat{L}([m_v]_M)$$

and finally any return values are copied back:

$$[m_v]_M.\mathit{returnVal} \Rightarrow \hat{S}([m_v]_M) \subseteq \hat{S}([m]_M)$$

In Figure 6.4 the full Flow Logic specification is shown.

For programs the specification is extended to handle the sharing introduced in initial configurations:

$$
\begin{array}{l}
(\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} P \quad \text{iff} \\
\quad \forall (m,pc) \in P.\mathit{addresses}: \\
\qquad m.\mathit{instructionAt}(pc) = \texttt{instr} \Rightarrow (\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} (m,pc) : \texttt{instr} \\
\quad \forall \sigma \in P.\mathit{main}: \\
\qquad m_\sigma = \sigma.\mathit{entry} \Rightarrow [\mathsf{Ref}\ \sigma]_C \in \hat{L}([m_\sigma]_M) \\
\qquad \sigma.\mathit{sharing} = \sigma_1 :: \cdots :: \sigma_n \Rightarrow \{[\mathsf{Ref}\ \sigma_1]_C, \ldots, [\mathsf{Ref}\ \sigma_n]_C\} \subseteq \hat{L}([m_\sigma]_M)
\end{array}
$$

### 6.3.3 Semantic Correctness

Proving the semantic correctness of the *devil's advocate* analysis follows the pattern of the previous analyses: first representation functions and correctness relations are defined; these are then used to establish a subject reduction property for the analysis.

All numbers are represented by the same abstract value:

$$\beta_{\mathsf{DA,Num}}(n) = \{\mathsf{INT}\} \qquad \text{for } n \in \mathsf{Num}$$

Carmel Core does not support arrays and so only object references need be represented:

$$\beta^H_{\mathsf{DA,Ref}}(loc) = \{[\mathsf{Ref}\ \sigma]_C\} \qquad \text{if } H(loc).\mathit{class} = \sigma$$

Basic values in Carmel Core comprises numbers and object references:

$$\beta_{\mathsf{DA,Val}}(v) = \begin{cases} \beta_{\mathsf{DA,Num}}(v) & \text{if } v \in \mathsf{Num} \\ \beta_{\mathsf{DA,Ref}}(v) & \text{if } v \in \mathsf{Ref} \end{cases}$$

As a consequence of the simple abstract domains objects are represented as a single abstract value, namely the union of all its fields:

$$\beta^H_{\mathsf{DA,Object}}(o) = \bigcup_{f \in \mathrm{dom}(o)} \beta^H_{\mathsf{DA,Val}}(o.f)$$

Local heaps are also represented as the union of the value of all the local variables:

$$\beta^H_{\mathsf{DA,LocHeap}}(L) = \bigcup_{x \in \mathrm{dom}(L)} \beta^H_{\mathsf{DA,Val}}(L(x))$$

Likewise, all stack positions are merged:

$$\beta^H_{\mathsf{DA,Stack}}(S) = \bigcup_{1 \le i \le |S|} \beta^H_{\mathsf{DA,Val}}(S|_i)$$

The representation function for heaps is very similar to previous definitions:

$$\beta_{\mathsf{DA,Heap}}(H)([\mathsf{Ref}\ \sigma]_C) = \bigcup_{\substack{loc \in \mathrm{dom}(H) \\ \beta^H_{\mathsf{DA,Val}}(loc) = [\mathsf{Ref}\ \sigma]_C}} \beta^H_{\mathsf{DA,Object}}(H(loc))$$

With the above representation functions in place the correctness relations are easily defined and contain no surprises:

$$\begin{aligned}
(m, pc, L, S)\ \mathcal{R}^{H}_{\mathsf{DA,Frame}}\ (\hat{L}, \hat{S}) \quad &\text{iff} \quad \beta^{H}_{\mathsf{DA,LocHeap}}(L) \sqsubseteq \hat{L}([m]_M) \wedge \\
&\qquad \beta^{H}_{\mathsf{DA,Stack}}(S) \sqsubseteq \hat{S}([m]_M) \\
SF\ \mathcal{R}^{H}_{\mathsf{DA,CallStack}}\ (\hat{L}, \hat{S}) \quad &\text{iff} \quad \forall i \in \{1, \ldots, |SF|\} : SF|_i\ \mathcal{R}^{H}_{\mathsf{DA,Frame}}\ (\hat{L}, \hat{S}) \\
\langle H, SF\rangle\ \mathcal{R}_{\mathsf{DA,Conf}}\ (\hat{H}, \hat{S}, \hat{L}) \quad &\text{iff} \quad \beta_{\mathsf{DA,Heap}}(H) \sqsubseteq \hat{H} \wedge \\
&\qquad SF\ \mathcal{R}^{H}_{\mathsf{DA,CallStack}}\ (\hat{L}, \hat{S})
\end{aligned}$$

This then allows us to state and prove the soundness of the analysis:

**Theorem 6.9 (Soundness).** *If* $P \in \mathsf{Program}$ *such that* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} P$*, and* $C_1, C_2 \in \mathsf{Conf}$ *with* $P \vdash C_1 \Longrightarrow C_2$ *then*

$$C_1\ \mathcal{R}_{\mathsf{DA,Conf}}\ (\hat{H}, \hat{S}, \hat{L}) \Rightarrow C_2\ \mathcal{R}_{\mathsf{DA,Conf}}\ (\hat{H}, \hat{S}, \hat{L})$$

*Proof.* Trivial adaptation of the proof for Theorem 3.14. ∎

Note that for the proof Lemma 3.11 can be re-used without modification since it trivially covers Carmel Core.

As a simple consequence of the above theorem it is possible to prove the corollary below showing that the control flow analysis can be used to guarantee that a given program does not leak references. For notational convenience define $[m]_M.\mathit{class} = \{[\sigma]_C \mid m' \in [m]_M, \sigma \in m'.\mathit{class}\}$:

**Corollary 6.10.** *Let* $P \in \mathsf{Program}$ *such that* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} P$*. Assume that* $[\mathsf{Ref}\ \tau]_C \notin \hat{H}([\mathsf{Ref}\ \sigma]_C)$ *and for all* $[m_v]_M$ *such that* $[\sigma]_C \in [m_v]_M.\mathit{class}$ *it is the case that* $[\mathsf{Ref}\ \tau]_C \notin (\hat{S}([m_v]_M) \cup \hat{L}([m_v]_M))$ *then* $P \vdash \tau \not\rightsquigarrow \sigma$*.*

The corollary shows that if an abstract reference to $\tau$ is not found in any heap location belonging to $\sigma$ or in any local heap or operand stack of a method that is a member of $\sigma$, then $\tau$ is not leaked to $\sigma$.

**Example 6.11.** *Using the control flow analysis on the example programs* $P_{ABM}$ *and* $P_{ABC}$ *results in analyses such that* $(\hat{H}_{ABM}, \hat{S}_{ABM}, \hat{L}_{ABM}) \models_{DA} P_{ABM}$ *and* $(\hat{H}_{ABC}, \hat{S}_{ABC}, \hat{L}_{ABC}) \models_{DA} P_{ABC}$*. Below the analysis results for the classes* `Mallet` *and* `Charlie` *are shown. The corresponding results for* `Alice` *and* `Bob` *are of no consequence, cf. Corollary 6.10:*

$$\begin{aligned}
\hat{S}_{ABM}([\texttt{Mallet.m_Mallet}]_M) &= \{[\mathsf{Ref}\ \texttt{Alice}]_C, [\mathsf{Ref}\ \texttt{Bob}]_C, [\mathsf{Ref}\ \texttt{Mallet}]_C\} \\
\hat{L}_{ABM}([\texttt{Mallet.m_Mallet}]_M) &= \{[\mathsf{Ref}\ \texttt{Bob}]_C, [\mathsf{Ref}\ \texttt{Mallet}]_C\} \\
\hat{H}([\mathsf{Ref}\ \texttt{Mallet}]_C) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\hat{S}_{ABC}([\texttt{Charlie.m_Charlie}]_M) &= \{[\mathsf{Ref}\ \texttt{Bob}]_C, [\mathsf{Ref}\ \texttt{Charlie}]_C\} \\
\hat{L}_{ABC}([\texttt{Charlie.m_Charlie}]_M) &= \{[\mathsf{Ref}\ \texttt{Bob}]_C, [\mathsf{Ref}\ \texttt{Charlie}]_C\} \\
\hat{H}([\mathsf{Ref}\ \texttt{Charlie}]_C) &= \emptyset
\end{aligned}$$

*Since* $[\mathsf{Ref}\ \texttt{Alice}]_C \in \hat{S}([\texttt{Mallet.m\_Mallet}]_M)$ *it can be deduced that* possibly $P_{ABM} \vdash \texttt{Alice} \rightsquigarrow \texttt{Mallet}$ *which is consistent with earlier observations. On the other hand Corollary 6.10 can be used to conclude that* $P_{ABC} \vdash \texttt{Alice} \not\rightsquigarrow \texttt{Charlie}$.

## 6.4 The Devil's Advocate

In this section a solution is presented to a problem posed earlier: how to guarantee that a given applet does not leak object references, regardless of what other malicious applets run in the same environment. In Section 6.2 it was discussed how the control flow analysis could be used to verify that a given set of applets running in a given environment does not leak references, which is not sufficient to deal with applets that are loaded after this analysis has been performed. Thus a different approach is needed. Following [Han04, NNH02, NN00] the *devil's advocate* (or *hardest attacker*) with respect to the control flow analysis defined in this chapter is identified. The devil's advocate is an attacker with the property that is no less "effective" than any other attacker and therefore, if the devil's advocate cannot perpetrate a successful attack then *no other attacker* can. Rather than try and find a finite characterisation of the infinitely many possible attackers (possibly in the form of a "universal" Carmel program) the key observation is that it only needs to be a devil's advocate relative to the control flow analysis. This has the major advantage that irrelevant details in the semantics can be abstracted away and thereby greatly facilitate the construction of the devil's advocate.

The basic idea behind the devil's advocate presented here is that, modulo names, a program only gives rise to finitely many "types" of Flow Logic clauses as determined by the instructions of the program. It is therefore possible to specify a finite set of clauses that contains or implies all the clauses a given program may give rise to (again modulo names). Such a set of clauses is then called the devil's advocate. Since the clauses do not rely on particular names that are used, but rather on the equivalence classes of names, the equivalence relations on classes and methods can be used to handle the possibly infinitely many names used in an attacker by simply choosing an equivalence relation that partitions names into finitely many equivalence classes.

In order to be able to define when an object reference is leaked, it is necessary to define a *sharing policy* that specifies which classes are allowed to be shared and which classes (and methods) that should not be shared (or leaked):

**Definition 6.12 (Sharing Policy).** *A* sharing policy *is a map,* $\mathcal{S}\colon \mathsf{Class} \cup \mathsf{Method} \to \{\mathsf{private}, \mathsf{shareable}, \mathsf{public}\}$ *that partitions classes and methods into private, shareable, and public in such a way that*

$$\forall \sigma \in \mathsf{Class} : \forall m \in \sigma.methods : \mathcal{S}(m) = \mathsf{public} \Rightarrow \mathcal{S}(\sigma) = \mathsf{public}$$

Thus, a sharing policy does not allow public classes to contain (or inherit) non-public methods. The restriction on inherited methods can be removed by using

*static inheritance* where inherited methods are syntactically duplicated in the inheriting class instead of being looked up dynamically.

A program, $P \in \mathsf{Program}$, that only defines and creates public classes and methods is called a *public program*:

**Definition 6.13 (Public Program).** *Given a sharing policy, $\mathcal{S}$, a program, $P \in \mathsf{Program}$, is said to be a* public program *with respect to $\mathcal{S}$ if the following holds:*

1. $\forall \sigma \in P.classes : \mathcal{S}(\sigma) = \mathsf{public}$
2. $\forall m \in P.methods : \mathcal{S}(m) = \mathsf{public}$
3. $\forall (m, pc) \in P.addresses : m.instructionAt(pc) = \texttt{new}\ \sigma \Rightarrow \mathcal{S}(\sigma) = \mathsf{public}$
4. $\forall \sigma \in P.main : \sigma.sharing = \sigma_1 :: \cdots :: \sigma_n \Rightarrow \forall i \in 1, \ldots, n : \mathcal{S}(\sigma_i) = \mathsf{shareable}$

*The set of public programs with respect to $\mathcal{S}$ is denoted* $\mathsf{Program}^{\mathcal{S}}_{\bullet}$.

A special equivalence relation, called the *discrete equivalence*, can now be defined with the intention of partitioning the names in a program in accordance with the sharing policy:

**Definition 6.14 (Discrete Equivalence).** *Let $P \in \mathsf{Program}$ and $\mathcal{S}$ be a sharing policy. The* discrete equivalence *relations on classes and methods, written $[\cdot]^{\mathcal{S}}_C$ and $[\cdot]^{\mathcal{S}}_M$ respectively, is then defined as follows*

$$\begin{array}{rcl}
\forall \sigma \in \mathsf{Class} : \mathcal{S}(\sigma) = \mathsf{private} & \Rightarrow & [\sigma]^{\mathcal{S}}_C = \{\sigma\} \\
\forall \sigma \in \mathsf{Class} : \mathcal{S}(\sigma) = \mathsf{shareable} & \Rightarrow & [\sigma]^{\mathcal{S}}_C = \{\sigma\} \\
\forall \sigma \in \mathsf{Class} : \mathcal{S}(\sigma) = \mathsf{public} & \Rightarrow & [\sigma]^{\mathcal{S}}_C = \bullet_C \\
 & & \\
\forall m \in \mathsf{Method} : \mathcal{S}(\sigma) = \mathsf{private} & \Rightarrow & [m]^{\mathcal{S}}_M = \{\sigma\} \\
\forall m \in \mathsf{Method} : \mathcal{S}(\sigma) = \mathsf{shareable} & \Rightarrow & [m]^{\mathcal{S}}_M = \{\sigma\} \\
\forall m \in \mathsf{Method} : \mathcal{S}(\sigma) = \mathsf{public} & \Rightarrow & [m]^{\mathcal{S}}_M = \bullet_M
\end{array}$$

The following lemma characterises the interaction of method lookups with program extensions:

**Lemma 6.15.** *Given $Q \in \mathsf{Program}^{\mathcal{S}}_{\bullet}$ for a sharing policy $\mathcal{S}$. Then the following holds:*

$$\begin{array}{l}
\forall m \in \mathsf{Method}: : \forall \sigma \in Q.classes: \\
\quad methodLookup_{/\equiv}(m, [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) = \{\bullet_M\}
\end{array}$$

*Proof.* Follows from the fact that $Q$ is a public program and therefore, by definition, does not define or inherit non-public methods. Any method looked up in a (public) class of $Q$ must therefore be public. ∎

Based on the above lemma it is now possible to entirely remove the dependence on actual method names in the analysis by replacing $methodLookup_{/\equiv}$ with a conservative approximation of it:

$$\widehat{methodLookup}_{/\equiv}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) = \begin{cases} \bigcup_{m\,\in\,P.methods} methodLookup_{/\equiv}(m, [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) & \text{if } [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C \neq \bullet_C \\ \{\bullet_M\} & \text{otherwise} \end{cases}$$

The abstract method lookup simply merges all the methods of all the classes belonging to the equivalence class for a given object reference. In other words: a method lookup for a given reference will return a conservative estimate of all the methods accessible through the given object reference, regardless of the method names and thus the dependency on method names is dispensed with. This approach is sufficient for the intended use, where a private program is extended with a public program, because a public program can only define (and inherit) public methods. It is possible to accommodate a more relaxed notion of public programs that are allowed to inherit (but not define) non-public methods. A more general, and precise, alternative is to completely dispense with the dynamic method lookup and replace it by *static inheritance*, i.e., syntactically duplicating inherited method definitions and thereby obviating the need for the $methodLookup_{/\equiv}$ function. Using static inheritance would, however, require changing the source program which may not be acceptable in all situations.

The following lemma formalises that the abstract method lookup is a sound approximation of the concrete method lookup used in the semantics:

**Lemma 6.16.** *Let $\mathcal{S}$ be a sharing policy and $[\cdot]^{\mathcal{S}}_C$ and $[\cdot]^{\mathcal{S}}_M$, the corresponding discrete equivalence. Then for all $\sigma \in \mathsf{Class}$ and $m_0 \in \mathsf{Method}$*

$$\forall [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C : \forall m_0 : \\ methodLookup_{/\equiv}(m_0, [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) \subseteq \widehat{methodLookup}_{/\equiv}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C)$$

*Proof.* For all $\sigma$ such that $\mathcal{S}(\sigma) \neq \mathsf{public}$ it is the case that $[\mathsf{Ref}\ \sigma]_C \neq \bullet_C$ and the result follows trivially from the definition. For $\sigma$ with $\mathcal{S}(\sigma) = \mathsf{public}$ the class $\sigma$, by definition of a public class, does not contain any non-public methods and the result follows, cf. Lemma 6.15. ∎

This leads to the formal definition of the devil's advocate below. Note how the individual clauses are very similar to the corresponding clauses for the analysis. This suggests the possibility that the construction of a devil's advocate for a given analysis can be done in a systematic, if not automatic, manner.

**Definition 6.17 (The Devil's Advocate).** *Let $P \in \mathsf{Program}$ and $\mathcal{S}$ a sharing policy and let $[\cdot]^{\mathcal{S}}_C$ and $[\cdot]^{\mathcal{S}}_M$ be the discrete equivalence. Then the* Devil's

Advocate *with respect to* $P$ *and* $\mathcal{S}$ *is then defined as:*

$$
\begin{array}{l}
(\hat{H},\hat{S},\hat{L}) \models_{DA} \mathcal{D}_P^{\mathcal{S}} \\
\quad \textit{iff} \quad \{\bullet_C\} \subseteq \hat{S}(\bullet_M) \wedge \\
\qquad \hat{L}(\bullet_M) \subseteq \hat{S}(\bullet_M) \wedge \\
\qquad \hat{S}(\bullet_M) \subseteq \hat{L}(\bullet_M) \wedge \\
\qquad \{\mathsf{INT}\} \subseteq \hat{S}(\bullet_M) \wedge \\
\qquad \forall [\mathsf{Ref}\ \sigma]_C^{\mathcal{S}} \in \hat{S}(\bullet_M) : \hat{H}([\mathsf{Ref}\ \sigma]_C^{\mathcal{S}}) \subseteq \hat{S}(\bullet_M) \wedge \\
\qquad \forall [\mathsf{Ref}\ \sigma]_C^{\mathcal{S}} \in \hat{S}(\bullet_M) : \hat{S}(\bullet_M) \subseteq \hat{H}([\mathsf{Ref}\ \sigma]_C^{\mathcal{S}}) \wedge \\
\qquad \forall \sigma \in P.main : \mathcal{S}(\sigma) = \mathsf{shareable} \Rightarrow \left\{[\mathsf{Ref}\ \sigma]_C^{\mathcal{S}}\right\} \subseteq \hat{L}(\bullet_M) \wedge \\
\qquad \forall [\mathsf{Ref}\ \sigma]_C^{\mathcal{S}} \in \hat{S}(\bullet_M) : \\
\qquad\quad \forall [m_v]_M^{\mathcal{S}} \in \overbrace{methodLookup}_{/\equiv}([\mathsf{Ref}\ \sigma]_C^{\mathcal{S}}) \\
\qquad\qquad \hat{S}(\bullet_M) \subseteq \hat{L}([\mathsf{Ref}\ \sigma]_C^{\mathcal{S}}) \\
\qquad\qquad [m_v]_M^{\mathcal{S}}.returnVal \Rightarrow \hat{S}([m_v]_M^{\mathcal{S}}) \subseteq \hat{S}(\bullet_M)
\end{array}
$$

The following lemma establishes that the devil's advocate defines an acceptable analysis for *all* public programs:

**Lemma 6.18.** *Let* $P \in \mathsf{Program}$ *and let* $\mathcal{S}$ *be a sharing policy. Then* $\forall Q \in \mathsf{Program}_\bullet^{\mathcal{S}} : (\hat{H},\hat{S},\hat{L}) \models_{DA} \mathcal{D}_Q^{\mathcal{S}} \Rightarrow (\hat{H},\hat{S},\hat{L}) \models_{DA} Q$.

*Proof.* By definition of $(\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} Q$ it must be shown that:

$$
\begin{array}{l}
\forall \sigma \in Q.main : \\
\quad m_\sigma = \sigma.entry \Rightarrow [\mathsf{Ref}\ \sigma]_C \in \hat{L}([m_\sigma]_M) \\
\quad \sigma.sharing = \sigma_1 :: \cdots :: \sigma_n \Rightarrow \{[\mathsf{Ref}\ \sigma_1]_C, \ldots, [\mathsf{Ref}\ \sigma_n]_C\} \subseteq \hat{L}([m_\sigma]_M)
\end{array}
\tag{6.3}
$$

and

$$
\begin{array}{l}
\forall (m,pc) \in Q.addresses : \\
\quad m.instructionAt(pc) = \texttt{instr} \Rightarrow (\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} (m,pc) : \texttt{instr}
\end{array}
\tag{6.4}
$$

However, since $Q$ is a public program it is the case that for all $\sigma \in Q.classes$ : $[\mathsf{Ref}\ \sigma]_C = \bullet_C$ and $[m_\sigma]_M = \bullet_M$. Furthermore for all the $\sigma_i$ it holds that $\mathcal{S}(\sigma_i) = \mathsf{shareable}$ and thus equation (6.3) is equivalent to

$$
\begin{array}{l}
\forall \sigma \in Q.main : \\
\quad m_\sigma = \sigma.entry \Rightarrow \{\bullet_C\} \subseteq \hat{L}(\bullet_M) \\
\quad \sigma.sharing = \sigma_1 :: \cdots :: \sigma_n \Rightarrow \{[\mathsf{Ref}\ \sigma_1]_C, \ldots, [\mathsf{Ref}\ \sigma_n]_C\} \subseteq \hat{L}(\bullet_M)
\end{array}
$$

which easily follows from $(\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} \mathcal{D}_Q^{\mathcal{S}}$.

The requirements of equation (6.4) are shown by case analysis on the instruction in $Q$:

*Case* `new`*:* Assume that $(m,pc) \in Q.addresses : m.instructionAt(pc) = \texttt{new}\ \sigma$. Since $Q$ is a public program it follows from Definition 6.13 that $\mathcal{S}(m) = \mathsf{public}$ and $\mathcal{S}(\sigma) = \mathsf{public}$, thus thus $[m]_M^{\mathcal{S}} = \bullet_M$ and $[\sigma]_C^{\mathcal{S}} = \bullet_C$. From $(\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} \mathcal{D}_Q^{\mathcal{S}}$ it follows that $\{\bullet_C\} \subseteq \hat{S}(\bullet_M)$. In combination this implies that $(\hat{H},\hat{S},\hat{L}) \models_{\mathrm{DA}} (m,pc) : \texttt{new}\ \sigma$.

*Case* `invokevirtual`*:* Assume $m.instructionAt(pc) = \texttt{invokevirtual}\ m_0$ for some $(m, pc) \in Q.addresses$ . Since $Q$ is a public program it follows that $[m]^{\mathcal{S}}_M = \bullet_M$. From $(\hat{H}, \hat{S}, \hat{L}) \models_{\text{DA}} \mathcal{D}^{\mathcal{S}}_Q$ it follows that

$$\begin{array}{l}\forall [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C \in \hat{S}(\bullet_M) : \\ \quad \forall [m_v]^{\mathcal{S}}_M \in \widehat{methodLookup}_{/\equiv}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) : \\ \quad\quad \hat{S}(\bullet_M) \subseteq \hat{L}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) \\ \quad\quad [m_v]^{\mathcal{S}}_M.returnVal \Rightarrow \hat{S}([m_v]^{\mathcal{S}}_M) \subseteq \hat{S}(\bullet_M)\end{array}$$

Using Lemma 6.16 the above implies

$$\begin{array}{l}\forall [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C \in \hat{S}(\bullet_M) : \\ \quad \forall [m_v]^{\mathcal{S}}_M \in methodLookup_{/\equiv}(m_0, [\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) : \\ \quad\quad \hat{S}(\bullet_M) \subseteq \hat{L}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C) \\ \quad\quad [m_v]^{\mathcal{S}}_M.returnVal \Rightarrow \hat{S}([m_v]^{\mathcal{S}}_M) \subseteq \hat{S}(\bullet_M)\end{array}$$

which is equivalent to $(\hat{H}, \hat{S}, \hat{L}) \models_{\text{DA}} \texttt{invokevirtual}\ m_0$.

The remaining cases are analogous or trivial. ∎

The main theorem for the devil's advocate analysis can now be stated and proved:

**Theorem 6.19.** *Let* $P \in \mathsf{Program}$ *and* $\mathcal{S}$ *a sharing policy then* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} P$ *and* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} \mathcal{D}_{\mathcal{S}}$ *implies that* $\forall Q \in \mathsf{Program}^{\mathcal{S}}_{\bullet} : (\hat{H}, \hat{S}, \hat{L}) \models_{DA} P\langle Q \rangle$.

*Proof.* Follows from lemmas 6.16 and 6.18. ∎

It is an immediate corollary that the control flow analysis and the devil's advocate can be used to verify that a (non-public) program does not leak certain references to any public program:

**Corollary 6.20.** *Let* $P \in \mathsf{Program}$ *and let* $\mathcal{S}$ *be a sharing policy such that* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} P$ *and* $(\hat{H}, \hat{S}, \hat{L}) \models_{DA} \mathcal{D}_{\mathcal{S}}$. *Assume that* $[\mathsf{Ref}\ \tau]^{\mathcal{S}}_C \notin \hat{H}([\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C)$ *and for all* $[m]^{\mathcal{S}}_M$ *such that* $[\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C \in [m]^{\mathcal{S}}_M.class$ *implies that* $[\mathsf{Ref}\ \sigma]^{\mathcal{S}}_C \notin (\hat{S}([m]^{\mathcal{S}}_M) \cup \hat{L}([m]^{\mathcal{S}}_M))$ *then* $\forall Q \in \mathsf{Program}^{\mathcal{S}}_{\bullet} : P\langle Q \rangle \vdash \tau \not\rightsquigarrow \sigma$.

In other words: if $\tau$ is not leaked to $\sigma$ when analysing $P$ in conjunction with the devil's advocate, then $P$ will *never* leak $\tau$ to $\sigma$ in *any* public program $Q$.

**Example 6.21.** *In Figure 6.5 an analysis result,* $(\hat{H}_{AB}, \hat{S}_{AB}, \hat{L}_{AB})$*, for the example program* $P_{AB}$*, i.e.,* $(\hat{H}_{AB}, \hat{S}_{AB}, \hat{L}_{AB}) \models_{DA} P_{AB}$*. As can be seen the analysis is very imprecise. However, the results do indicate that the program* may *leak an* `Alice`*-reference to a public program:*

$$[\mathsf{Ref}\ \texttt{Alice}]^{\mathcal{S}}_C \in (\hat{S}(\bullet_M) \cap \hat{L}(\bullet_M) \cap \hat{H}(\bullet_C))$$

*This is consistent with earlier observations.*

$$
\begin{array}{rcl}
\hat{S}_{AB}([\texttt{Alice.m\_Alice}]^{\mathcal{S}}_{M}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{S}_{AB}([\texttt{Bob.update}]^{\mathcal{S}}_{M}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{S}_{AB}(\bullet_M) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
 & & \\
\hat{L}_{AB}([\texttt{Alice.m\_Alice}]^{\mathcal{S}}_{M}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{L}_{AB}([\texttt{Bob.m\_Bob}]^{\mathcal{S}}_{M}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{L}_{AB}([\texttt{Bob.update}]^{\mathcal{S}}_{M}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{L}_{AB}(\bullet_M) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
 & & \\
\hat{H}_{AB}([\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{H}_{AB}([\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\} \\
\hat{H}_{AB}(\bullet_C) & = & \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref } \texttt{Bob}]^{\mathcal{S}}_{C}, \bullet_C\right\}
\end{array}
$$

Figure 6.5: Analysis result for $P_{AB}$ in conjunction with the devil's advocate

```
class Alice {
  void m_Alice(Bob) {
    0: load r 1
    1: load r 0
    2: invokevirtual Bob.update(Object)
    3: return
  }
  /* ... */
}
```

```
class Bob {
  void m_Bob() {
    0: return
  }
  void update(Object) {
    /* ... */
    0: return
  }
}
```

Figure 6.6: Program $P_{AB'}$: a corrected version of $P_{AB}$

**Example 6.22.** *In Figure 6.6 a corrected version of program $P_{AB}$ is shown. The* `update` *method of the* `Bob` *class no longer caches the* `Alice`*-reference.*

*The corresponding analysis result, $(\hat{H}_{AB'}, \hat{S}_{AB'}, \hat{L}_{AB'}) \models_{DA} P_{AB'}$, for the corrected program in conjunction with the devil's advocate can be seen in Figure 6.7. The main feature of the analysis result is that no* `Alice`*-reference can be leaked to a public program:*

$$(\hat{S}(\bullet_M) \cup \hat{L}(\bullet_M) \cup \hat{H}(\bullet_C)) \cap \left\{[\textsf{Ref } \texttt{Alice}]^{\mathcal{S}}_{C}\right\} = \emptyset$$

### 6.4.1 Implementation

Both the simplified control flow analysis and the devil's advocate has been implemented in a prototype analysis and verification tool. The simplified control flow analysis was implemented by adding relevant lookups (into encodings of the equivalence relations) to the implementation of the "normal" control flow

$$
\begin{array}{rcl}
\hat{S}_{AB'}([\texttt{Alice.m\_Alice}]^{\mathcal{S}}_{M}) & = & \{[\textsf{Ref Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref Bob}]^{\mathcal{S}}_{C}\} \\
\hat{S}_{AB'}([\texttt{Bob.update}]^{\mathcal{S}}_{M}) & = & \{[\textsf{Ref Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
\hat{S}_{AB'}(\bullet_M) & = & \{[\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
 & & \\
\hat{L}_{AB'}([\texttt{Alice.m\_Alice}]^{\mathcal{S}}_{M}) & = & \{[\textsf{Ref Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref Bob}]^{\mathcal{S}}_{C}\} \\
\hat{L}_{AB'}([\texttt{Bob.m\_Bob}]^{\mathcal{S}}_{M}) & = & \{[\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
\hat{L}_{AB'}([\texttt{Bob.update}]^{\mathcal{S}}_{M}) & = & \{[\textsf{Ref Alice}]^{\mathcal{S}}_{C}, [\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
\hat{L}_{AB'}(\bullet_M) & = & \{[\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
 & & \\
\hat{H}_{AB'}([\textsf{Ref Bob}]^{\mathcal{S}}_{C}) & = & \{[\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\} \\
\hat{H}_{AB'}(\bullet_C) & = & \{[\textsf{Ref Bob}]^{\mathcal{S}}_{C}, \bullet_C\}
\end{array}
$$

Figure 6.7: Analysis result for $P_{AB'}$ and the devil's advocate

analysis described in Section 3.4. The equivalence relations were encoded "manually" for each example program. The prototype implementation was used to obtain the analysis result in Example 6.22.

## 6.5 Summary

In this chapter the *devil's advocate* was developed for a simplified control flow analysis and was shown to adequately simulate the behaviour of all public programs and in particular all dynamically downloaded applets. The devil's advocate was used to verify that certain sensitive object references were not leaked to *any* public program including any applets loaded at a later stage. The general technique of defining a devil's advocate can be adapted to other Flow Logic based analyses and properties in a straightforward manner.

In [BGH02] a proof system is developed in which it is possible to decompose a global security property (or basically any property that can be formulated in the modal $\mu$-calculus) into local properties that must be proved (or model checked) for the individual applets resulting in a compositional proof system capable of handling dynamically downloaded applets. The local properties can be model checked whereas the proof of correct decomposition must be proved by hand. These ideas are further studied in [HGSC04] where the notion of *maximal applet* is introduced as a way to lift some of the restrictions of the earlier paper, in particular it is now possible to automatically very (by model checking) that the decomposition is correct thus allowing a fully automated verification. The maximal applet is conceptually very similar to the devil's advocate albeit in a rather different setting (model checking).

A different approach to solving a related problem, library analysis, is taken in [BBFG04]. Here an *access control sensitive* control flow analysis for the Common Language Runtime (CLR) is developed. The analysis is used to guarantee

that no untrusted code can ever obtain sufficient runtime permissions to reach certain sensitive methods directly. In this approach there is no equivalent of a devil's advocate; it is instead an intrinsic part of the analysis which makes it difficult to assess how hard it would be to adapt the technique to other analyses and properties.

Chapter 7

# Conclusions

*You can't trust code that you did not totally create yourself.*
*(Especially code from companies that employ people like me.)*
*—Ken Thompson in [Tho84]*

In the preceding chapters the Carmel language, a low-level stack-based Java Card bytecode-like language, was introduced and its semantics formally specified. A number of analyses were developed and formally proved correct with respect to the semantics and then used as basis for statically verifying and validating a selection of safety and security properties. In the remainder of the chapter related work is briefly discussed and it is argued how this work supports the main thesis of the dissertation.

## 7.1 Related Work

In the summary section of each chapter the most relevant directly related work is discussed. Here the focus is on related general approaches to verifying safety and security properties.

The work in this dissertation grew from the SecSafe project whose goal was to investigate the use of static analysis to verification and validation of security properties of JCVML. See [Siv03b] for a comprehensive description and a publication list.

Java smart cards were also the target for the VerifiCard project, cf. [ver05]. The project mainly focused on using model checking and theorem proving rather than static analysis. However, the project verified the correctness of not only individual applets but also important components of the Java Card infrastructure, e.g., the bytecode verifier and the virtual machine. Both Java Card proper and JCVML were targeted in the VerifiCard project.

The KeY project, cf. [key05], uses a commercial CASE tool as a starting point. The tool is then enhanced with support for formal specification and verification. The tool for verification is a theorem prover based on a variant of *dynamic logic*, cf. [HKT00], specifically tailored to Java Card programs, cf. [Bec01]. The specialised dynamic logic is used to specify both an abstract model of the target applet and the property to be proved.

By annotating Java Card programs with extra information, e.g., preconditions and postconditions for methods, the Java Modelling Language (JML), cf. [JML05, BCC$^+$], enables external tools to verify a wide range of safety and security properties. Some tools use static analysis to automatically add and/or propagate relevant annotations, cf. [PBB$^+$04], and thus achieve a high degree of automation. The underlying idea of JML is quite similar to that of the SPARK approach [Bar03] which is based on a safe subset of Ada, SPARK Ada, rather than Java Card. The SPARK approach also combines the use of annotations, static analysis, and a simple theorem prover to develop and validate high-assurance applications.

The above represents only a small sample of the many projects that work with verification of some variant of Java. For a comprehensive overview of research in this field see [HM01].

The notion of language-based safety and security used in this dissertation falls within the definition used in [SMH01] and is influenced from the early work in this field, i.e., [VSI96, Mil81, DD77, Den76, AR80]. In recent years the notion of language-based security has come to include approaches based on other ideas from the programming language community, e.g., certifying compilers (in the form of proof-carrying code) [Nec97] and in-lined reference monitors [ES00, ES99].

The use of tools and techniques from programming language theory also raises the question of the formal limits for automating safety and security verification. An early paper [HRU76] proves that in general the problem of determining whether or not a system (modelled using the access matrix model) can end in an in-secure state is undecidable. More recently this has been studied in [HMS03, Sch00] using a model based on *security automata*.

## 7.2 Conclusions

The main thesis of the present work is that:

> *The Flow Logic framework for static analysis is a powerful tool for language-based safety and security.*

In order to support this thesis, the Flow Logic framework was examined with respect to the following five properties (or axes): versatility, flexibility, robustness, scalability, and implementability, cf. Chapter 1. In the following it is discussed how the Flow Logic framework exhibits the above properties.

**Versatility.** The non-standard language features of Carmel include subroutines, applet firewall, and transactions. All of which are supported within

the framework: subroutines as an integral part of the control flow analysis (Section 3.2), the applet firewall by the ownership analysis (Section 4.2), and transactions by the transaction flow analysis (Section 5.1).

**Flexibility.** While the flexibility of the framework is demonstrated by the wide range of properties covered by the analyses, the development of the devil's advocate in Chapter 6 further emphasises the usefulness of the framework as a platform for exploring novel properties and concepts.

**Robustness.** The development of the data flow analysis (Section 4.1) and the ownership analysis (Section 4.2) as simple extensions of the basic control flow analysis illustrates the robustness of the framework. It is even more evident for the proofs of correctness for the two analyses: both are realised as slightly modified versions of the proof for the control flow analysis. This allows analyses to be developed in a staged manner where additional language constructs or concepts can be added without having to start from scratch. The advanced analyses of Chapter 5 also re-use large parts of both analysis specification and proof of correctness for the earlier analyses further illustrating the robustness.

**Scalability.** The control flow analysis (Section 3.2) covers all of the standard features of Carmel including exceptions and subroutines. Non-standard features, e.g., ownership and transactions, are covered by specialised analyses. Thus the full Carmel (Carmel$_{\mathsf{EXC}}$ actually) language is covered by the analyses.

**Implementability.** Implementation issues have only been mentioned or discussed incidentally in this dissertation. However, most of the analyses have been implemented as prototypes by converting the abstract analysis specification into a constraint generator over ALFP, as described in Section 3.4, and then using the Succinct Solver to solve the constraints and obtain the analysis result.

The above is presented as evidence that the thesis is indeed correct. In addition to the main thesis the developments of this dissertation also illustrates the usefulness of static analysis for language-based safety and security and indeed of language-based safety and security in general. To support these additional propositions it is argued below how a large subset of the safety and security properties, identified in [MM01] as the relevant properties for smart card validation and verification from an industrial point of view, can be verified using the analyses developed in the previous chapters.

**Information Flow Control.** This property is directly targeted by the information flow analysis (Section 5.2).

**Service Control.** Using the control flow analysis (Section 3.2) and/or the exception analysis (Section 3.5) it is possible verify that certain instructions are only reachable if a specific set of conditionals are true.

**Error Prediction.** The exception analysis (Section 3.5) combined with the ownership analysis (Section 4.2) and the transaction flow analysis (Section 5.1) is sufficient to verify that only certain specified exceptions reach the top-level.

**Atomic Updates.** Using the transaction flow analysis (Section 5.1) to calculate the transaction depth of each instruction it is possible to verify that certain sets of operations are performed in the same transaction. It is relatively easy to obtain a more precise transaction flow analysis by tracking not only transaction depth but also place of origin for active transactions.

**Overflow Control.** The data flow analysis (Section 4.1) can be used to verify that numeric operations do not give rise to (silent) overflows. For numeric operations using arrays it may be necessary to enhance the data flow analysis with a more precise notion of abstract array.

The devil's advocate (Chapter 6) points to a way of extending verification based on static analysis to cope with dynamic environments such as smart cards and mobile code in general.

While there is little doubt that language-based techniques can and will play an essential part in the development of safe and secure systems for future computing paradigms, it is important to remember that language-based techniques are not a panacea. This is demonstrated very convincingly in [GA03] where the type safety of the Java virtual machine is attacked using a light-bulb to introduce runtime type-errors; a special attack-applet can then exploit the runtime type-error to gain unrestricted access to the entire memory.

Appendix A

# Carmel Semantics

## A.1 Full Semantics for Runtime Exceptions

### A.1.1 Imperative Core

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \{\texttt{div}, \texttt{rem}\} \qquad v_2 = 0 \qquad \sigma_X = \texttt{ArithmeticExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle K, S, F :: SF\rangle}$$

### A.1.2 Object Fragment

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{checkcast}\ \ \sigma \\ loc \neq \texttt{null} \Rightarrow H(loc).class \not\preceq \sigma \qquad \sigma_X = \texttt{ClassCastExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield}\ \ f \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield}\ \ f \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield this } f \\ L(0) = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield this } f \\ L(0) = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

### A.1.3 Method Fragment

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual } m_0 \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle \mathsf{Exc}\ loc_X\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokeinterface } m_0 \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{\begin{array}{r} P \vdash \langle K, H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \langle K, H, \langle \mathsf{Exc}\ loc_X\rangle :: SF\rangle \end{array}}$$

### A.1.4 Arrays

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{new (array } t\texttt{)} \\ n < 0 \qquad \sigma_X = \texttt{NegArraySizeExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, n :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{arraylength} \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{arrayload } t \\ \sigma_X = \begin{cases} \texttt{NullPointerExc} & \text{if } loc = \texttt{null} \\ \texttt{IndexOutOfBoundsExc} & \text{if } n < 0 \vee n \geq H(loc).length \end{cases} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, n :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{arraystore } t \\ \sigma_X = \begin{cases} \texttt{NullPointerExc} & \text{if } loc = \texttt{null} \\ \texttt{IndexOutOfBoundsExc} & \text{if } n < 0 \vee n \geq H(loc).length \end{cases} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, v :: n :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H', F :: SF\rangle}$$

### A.1.5 Exceptions

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{throw} \\ loc = \texttt{null} \qquad \sigma_X = \texttt{NullPointerExc} \\ loc_X = excLocation(H, \sigma_X) \qquad F = nextFrame(m, pc, L, S, loc_X, \sigma_X) \end{array}}{P \vdash \langle K, H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle K, H, F :: SF\rangle}$$

## A.2 Carmel Core Semantics

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ t\ \ n}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, c :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{pop}\ \ n}{\begin{array}{l} P \vdash \langle H, \langle m, pc, L, v_1 :: \cdots :: v_n :: S\rangle :: SF\rangle \Longrightarrow \\ \qquad\qquad \langle H, \langle m, pc+1, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{BinaryOp} \qquad v = op(v_1, v_2) \qquad op \in \{\texttt{div}, \texttt{rem}\} \Rightarrow v_2 \neq 0 \end{array}}{P \vdash \langle H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle S, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{numop}\ \ t\ \ op\ \ [t'] \\ op \in \mathsf{UnaryOp} \qquad v = op(v_1) \end{array}}{P \vdash \langle H, \langle m, pc, L, v_1 :: S\rangle :: SF\rangle \Longrightarrow \langle S, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{goto}\ \ pc_0}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc_0, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0 \\ pc_1 = \begin{cases} pc_0 & \text{if } cmp(v_1, v_2) = true \\ pc+1 & \text{otherwise} \end{cases} \end{array}}{P \vdash \langle H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc_1, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmp\ \ nul\ \ \texttt{goto}\ \ pc_0 \\ pc' = \begin{cases} pc_0 & \text{if } cmp(v_1, nul) = true \\ pc+1 & \text{otherwise} \end{cases} \end{array}}{P \vdash \langle H, \langle m, pc, L, v_1 :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc', L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{load}\ \ t\ \ x}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, L(x) :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{store}\ \ t\ \ x}{P \vdash \langle H, \langle m, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L[x \mapsto v], S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{new}\ \sigma \qquad \sigma \in \mathsf{Class} \qquad (loc, H') = newObject(\sigma, H)}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H', \langle m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{getfield}\ f \\ loc \neq \texttt{null} \qquad o = H(loc) \qquad v = o.fieldValue(f) \end{array}}{P \vdash \langle H, \langle m, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield}\ f \\ loc \neq \texttt{null} \qquad o = H(loc) \\ o' = o[fieldValue \mapsto o.fieldValue[f \mapsto v]] \end{array}}{\begin{array}{r} P \vdash \langle H, \langle m, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \langle H[loc \mapsto o'], \langle m, pc+1, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ m_0 \\ loc \neq \texttt{null} \qquad o = H(loc) \\ L_v = loc :: v_1 \cdots :: v_{|m_0|} m_v = methodLookup(m_0, o.class) \end{array}}{\begin{array}{r} P \vdash \langle H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \Longrightarrow \\ \langle H, \langle m_v, 0, L_v, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: loc :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{return} \\ S' = v'_1 :: \cdots :: v'_{|m|} :: loc :: S'' \end{array}}{\begin{array}{r} P \vdash \langle H, \langle m, pc, L, S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \\ \langle H, \langle m', pc'+1, L', S''\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{return}\ t \\ S' = v'_1 :: \cdots :: v'_{|m|} :: loc :: S'' \end{array}}{\begin{array}{r} P \vdash \langle H, \langle m, pc, L, v :: S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \\ \langle H, \langle m', pc'+1, L', v :: S''\rangle :: SF\rangle \end{array}}$$

Appendix B

# Constraint Generator

For convenience and ease of reference the two auxiliary predicates are repeated here:

$$\begin{aligned}&\mathsf{COPYSTACK}(m_0, pc_0, i, pc_1, j) \equiv \\ &\quad \forall x : \forall y : \forall v : \mathsf{SP}_{j-i}(x, y) \wedge \mathsf{S}(m_0, pc_0, x, v) \Rightarrow \mathsf{S}(m_0, pc_1, y, v)\end{aligned}$$

$$\begin{aligned}&\mathsf{COPYLOCHEAP}(m_0, pc_0, pc_1) \equiv \\ &\quad \forall x : \forall v : \mathsf{L}(m_0, pc_0, x, v) \Rightarrow \mathsf{L}(m_0, pc_1, x, v)\end{aligned}$$

$$\begin{aligned}&\mathcal{G}[\![(m, pc) : \texttt{nop}]\!] = \\ &\qquad \mathbf{1}\end{aligned}$$

$$\begin{aligned}&\mathcal{G}[\![(m, pc) : \texttt{push } t \ n]\!] = \\ &\qquad \mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \wedge \\ &\qquad \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1) \\ &\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)\end{aligned}$$

$$\begin{aligned}&\mathcal{G}[\![(m, pc) : \texttt{pop } n]\!] = \\ &\qquad \mathsf{COPYSTACK}(m, pc, n, pc+1, 0) \\ &\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)\end{aligned}$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{dup}\ \ i\ \ j]\!] \ = \\
\qquad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![0]\!], v) \\
\qquad \vdots \\
\qquad \forall v : \mathsf{S}(m, pc, [\![j-1]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![j-1]\!], v) \\
\qquad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![j]\!], v) \\
\qquad \vdots \\
\qquad \forall v : \mathsf{S}(m, pc, [\![i-1]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![i-1+j]\!], v) \\
\qquad \mathsf{COPYSTACK}(m, pc, j, pc+1, j+i) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{swap}\ \ i\ \ j]\!] \ = \\
\qquad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![j]\!], v) \\
\qquad \vdots \\
\qquad \forall v : \mathsf{S}(m, pc, [\![i-1]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![i-1+j]\!], v) \\
\qquad \forall v : \mathsf{S}(m, pc, [\![i]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![0]\!], v) \\
\qquad \vdots \\
\qquad \forall v : \mathsf{S}(m, pc, [\![j-1+i]\!], v) \ \Rightarrow\ \mathsf{S}(m, pc+1, [\![j-1]\!], v) \\
\qquad \mathsf{COPYSTACK}(m, pc, i+j, pc+1, i+j) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{numop}\ \ unop\ \ t\ \ [t']]\!] \ = \\
\qquad \mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \\
\qquad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 1) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{numop}\ \ binop\ \ t\ \ [t']]\!] \ = \\
\qquad \mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \\
\qquad \mathsf{COPYSTACK}(m, pc, 2, pc+1, 1) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{goto}\ \ pc_0]\!] \ = \\
\qquad \mathsf{COPYSTACK}(m, pc, 0, pc_0, 0) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc_0)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0]\!] \ = \\
\qquad \mathsf{COPYSTACK}(m, pc, 2, pc+1, 0) \\
\qquad \mathsf{COPYSTACK}(m, pc, 2, pc_0, 0) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc+1) \\
\qquad \mathsf{COPYLOCHEAP}(m, pc, pc_0)
\end{array}
$$

$\mathcal{G}[\![(m, pc) : \texttt{if}\ t\ cmp\ \texttt{goto}\ pc_0]\!] =$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc + 1, 0)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc_0, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc_0)$

$\mathcal{G}[\![(m, pc) : \texttt{lookupswitch}\ t\ (k_i\texttt{=>}pc_i)_1^n\ \texttt{default=>}pc_{n+1}]\!] =$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc_0, 0)$
$\quad \vdots$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc_n, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc_0)$
$\quad \vdots$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc_n)$

$\mathcal{G}[\![(m, pc) : \texttt{tableswitch}\ t\ l\texttt{=>}(pc_i)_0^n\ \texttt{default=>}pc_{n+1}]\!] =$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc_0, 0)$
$\quad \vdots$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc_{n+1}, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc_0)$
$\quad \vdots$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc_{n+1})$

$\mathcal{G}[\![(m, pc) : \texttt{load}\ t\ x]\!] =$
$\quad \forall v : \mathsf{L}(m, pc, x, v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc + 1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1)$

$\mathcal{G}[\![(m, pc) : \texttt{store}\ t\ x]\!] =$
$\quad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m, pc + 1, x, v) \wedge$
$\quad \forall y : \forall v : (y \neq x) \wedge \mathsf{L}(m, pc, y, v) \Rightarrow \mathsf{L}(m, pc + 1, y, v)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc + 1, 0)$

$\mathcal{G}[\![(m, pc) : \texttt{inc}\ t\ x\ c]\!] =$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc + 1, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1)$

$\mathcal{G}[\![(m, pc) : \texttt{new}\ \sigma]\!] =$
$\quad \mathsf{S}(m, pc + 1, [\![0]\!], (\mathsf{Ref}\ \sigma))$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc + 1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1)$

$\mathcal{G}[\![(m,pc) : \texttt{instanceof } \sigma]\!] =$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc+1, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{checkcast } \sigma]\!] =$
$\quad \mathsf{S}(m, pc+1, [\![0]\!], \mathsf{INT}) \wedge$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{getfield } f]\!] =$
$\quad \forall r : \forall v : \mathsf{S}(m, pc, [\![0]\!], r) \wedge \mathsf{H}(r, f, v) \Rightarrow \mathsf{S}(m, pc+1, [\![0]\!], v)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{getfield this } f]\!] =$
$\quad \forall r : \forall v :$
$\quad\quad \mathsf{L}(m, pc, 0, r) \wedge \mathsf{H}(r, f, v) \Rightarrow \mathsf{S}(m, pc+1, [\![0]\!], v)$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{putfield } f]\!] =$
$\quad \forall r : \forall v : \mathsf{S}(m, pc, [\![1]\!], r) \wedge \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{H}(r, f, v)$
$\quad \mathsf{COPYSTACK}(m, pc, 2, pc+1, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{putfield this } f]\!] =$
$\quad \forall r : \forall v : \mathsf{L}(m, pc, 0, r) \wedge \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{H}(r, f, v)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{getstatic } f]\!] =$
$\quad \forall v : \mathsf{K}(f, v) \Rightarrow \mathsf{S}(m, pc+1, [\![0]\!], v)$
$\quad \mathsf{COPYSTACK}(m, pc, 0, pc+1, 1)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{putstatic } f]\!] =$
$\quad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{K}(f, v)$
$\quad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 0)$
$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$

$\mathcal{G}[\![(m,pc) : \texttt{return}]\!] =$
$\quad \mathbf{1}$

$\mathcal{G}[\![(m,pc) : \texttt{return } t]\!] =$
$\quad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{S}(m, \mathsf{END}, [\![0]\!], v)$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{invokevirtual}\ m_0]\!] = \\
\quad \forall r : \forall m_v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \Rightarrow \mathsf{L}(m_v, 0, 0, r) \\
\quad \forall r : \forall m_v : \forall v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m_v, 0, 1, v) \\
\quad\quad \vdots \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![|m_0| - 1]\!], v) \Rightarrow \mathsf{L}(m_v, 0, |m_0|, v) \\
\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1) \\
\quad \text{if } m_0.returnType = \texttt{void} \text{ then} \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 0) \\
\quad \text{if } m_0.returnType \neq \texttt{void} \text{ then} \\
\quad\quad \forall r : \forall m_v : \forall v : \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m_v, \mathsf{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v) \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{invokeinterface}\ m_0]\!] = \\
\quad \forall r : \forall m_v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \Rightarrow \mathsf{L}(m_v, 0, 0, r) \\
\quad \forall r : \forall m_v : \forall v : \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m_v, 0, 1, v) \\
\quad\quad \vdots \\
\quad\quad \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m, pc, [\![|m_0| - 1]\!], v) \Rightarrow \mathsf{L}(m_v, 0, |m_0|, v) \\
\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1) \\
\quad \text{if } m_0.returnType = \texttt{void} \text{ then} \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 0) \\
\quad \text{if } m_0.returnType \neq \texttt{void} \text{ then} \\
\quad\quad \forall r : \forall m_v : \forall v : \mathsf{S}(m, pc, [\![|m_0|]\!], r) \wedge \mathsf{ML}(m, r, m_v) \wedge \\
\quad\quad\quad \mathsf{S}(m_v, \mathsf{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v) \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0| + 1, pc + 1, 1)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![(m,pc) : \texttt{invokestatic}\ m_0]\!] = \\
\quad \forall v : \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow \mathsf{L}(m_0, 0, 0, v) \\
\quad \vdots \\
\quad \forall v : \mathsf{S}(m, pc, [\![|m_0| - 1]\!], v) \Rightarrow \mathsf{L}(m_0, 0, (|m_0| - 1), v) \\
\quad \mathsf{COPYLOCHEAP}(m, pc, pc + 1) \\
\quad \text{if } m_0.returnType = \texttt{void} \text{ then} \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0|, pc + 1, 0) \\
\quad \text{if } m_0.returnType \neq \texttt{void} \text{ then} \\
\quad\quad \forall v : \mathsf{S}(m_0, \mathsf{END}, [\![0]\!], v) \Rightarrow \mathsf{S}(m, pc + 1, [\![0]\!], v) \\
\quad\quad \mathsf{COPYSTACK}(m, pc, |m_0|, pc + 1, 1)
\end{array}
$$

$$\mathcal{G}[\![(m, pc) : \texttt{new (array } t\texttt{)}]\!] =$$
$$\quad \mathsf{S}(m, pc+1, [\![0]\!], (\mathsf{Ref}\ (\texttt{array}\ t)))$$
$$\quad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 1)$$

$$\mathcal{G}[\![(m, pc) : \texttt{arraylength}]\!] =$$
$$\quad \mathsf{S}(m, pc, [\![0]\!], \mathsf{INT})$$
$$\quad \mathsf{COPYSTACK}(m, pc, 1, pc+1, 1)$$
$$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$$

$$\mathcal{G}[\![(m, pc) : \texttt{arrayload } t]\!] =$$
$$\quad \forall r : \forall v : \mathsf{S}(m, pc, [\![1]\!], r) \wedge \mathsf{H}(r, \texttt{ARRAY}, v) \Rightarrow$$
$$\quad\quad \mathsf{S}(m, pc+1, [\![0]\!], v)$$
$$\quad \mathsf{COPYSTACK}(m, pc, 2, pc+1, 1)$$
$$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$$

$$\mathcal{G}[\![(m, pc) : \texttt{arraystore } t]\!] =$$
$$\quad \forall r : \forall v : \mathsf{S}(m, pc, [\![2]\!], r) \wedge \mathsf{S}(m, pc, [\![0]\!], v) \Rightarrow$$
$$\quad\quad \mathsf{H}(r, \texttt{ARRAY}, v)$$
$$\quad \mathsf{COPYSTACK}(m, pc, 3, pc+1, 0)$$
$$\quad \mathsf{COPYLOCHEAP}(m, pc, pc+1)$$

Appendix C

# Transaction Flow Analysis

## C.1 Semantic Reduction Rules

$$\frac{m.instructionAt(pc) = \texttt{push}\ \ c}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc+1, L, n :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{pop}\ \ n}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, c :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{load}\ \ t\ \ x}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc+1, L, L(x) :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{store}\ \ t\ \ x}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau, m^{\tau_m}, pc+1, L[x \mapsto v], S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{numop}\ \ op \qquad v = v_1\ \ op\ \ v_2}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau, m^{\tau_m}, pc+1, L, v :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{goto}\ \ pc_0}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc_0, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmpOp\ \ \texttt{goto}\ \ pc_0 \qquad \neg cmp(v_1, v_2)}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau, m^{\tau_m}, pc+1, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{if}\ \ t\ \ cmp\ \ \texttt{goto}\ \ pc_0 \qquad cmpOp(v_1, v_2)}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc_0, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{new}\ \ \sigma \qquad \sigma \in \mathsf{Class} \qquad (loc, H') = newObject(\sigma, H)}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H', \langle \tau, m^{\tau_m}, pc + 1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{getfield}\ \ f \qquad v = H(loc).f}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau, m^{\tau_m}, pc + 1, L, v :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield}\ \ f \\ H' = H[loc \mapsto o'] \wedge o' = H(loc)[f \mapsto v] \end{array}}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H', \langle \tau, m^{\tau_m}, pc + 1, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ \ m' \\ S = v_1 :: \cdots :: v_{|m'|} :: loc :: S_0 \qquad m_v = methodLookup(m', o.class) \\ o = H(loc) \wedge L' = [0 \mapsto loc, 1 \mapsto v_1, \ldots, |m'| \mapsto v_{|m'|}] \end{array}}{\begin{array}{r} P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau, m_v^{\tau}, 0, L', \epsilon\rangle :: \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{return}}{\begin{array}{r} P \vdash \langle H, \langle \tau', m'^{\tau_{m'}}, pc', L', v :: S'\rangle :: \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \\ \langle H, \langle \tau', m^{\tau_m}, pc + 1, L, v :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{m.instructionAt(pc) = \texttt{API.getTransactionDepth}}{P \vdash \langle H, \langle \tau, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle \tau, m^{\tau_m}, pc + 1, L, \tau :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.beginTransaction}}{P \vdash \langle H, \langle 0, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 1, m^{\tau_m}, pc + 1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.commitTransaction}}{P \vdash \langle H, \langle 1, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 0, m^{\tau_m}, pc + 1, L, S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{API.abortTransaction}}{P \vdash \langle H, \langle 1, m^{\tau_m}, pc, L, S\rangle :: SF\rangle \Longrightarrow_{\text{TFA}} \langle H, \langle 0, m^{\tau_m}, pc + 1, L, S\rangle :: SF\rangle}$$

## C.2 Transaction Flow Analysis

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{push}\ t\ n$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\{n\} :: \hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{pop}\ n$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$A_1 :: \cdots :: A_n :: X \triangleleft \hat{S}_\delta(m, pc) :$
$X \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{load}\ t\ x$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\hat{L}_\delta(m, pc)(x) :: \hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{store}\ x$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$A :: X \triangleleft \hat{S}_\delta(m, pc) :$
$X \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$A \sqsubseteq \hat{L}_\delta(m, pc + 1)(x)$
$\hat{L}_\delta(m, pc) \sqsubseteq_{\{x\}} \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{numop}\ t\ \mathit{binop}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m, pc) :$
$\{\mathsf{INT}\} :: X \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{new}\ \sigma$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\{(\mathsf{Ref}\ \sigma)\} :: \hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, pc + 1)$

$(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{if}\ \ t\ \ cmpOp\ \ \texttt{goto}\ \ pc_0$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$A_1 :: A_2 :: X \triangleleft \hat{S}_\delta(m,pc) :$
$\widehat{cond}(cmpOp, A_1, A_2) \Rightarrow$
$\hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc_0)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc_0)$
$\hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc_0)$
$\widehat{cond}(\neg cmpOp, A_1, A_2) \Rightarrow$
$\hat{S}_\delta(m,pc) \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$
$\hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)$

$(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{getfield}\ \ f$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$B :: X \triangleleft \hat{S}_\delta(m,pc) :$
$\forall(\mathsf{Ref}\ \sigma') \in B : (\hat{H}(\mathsf{Ref}\ \sigma')(f)) :: X \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$
$\hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)$

$(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{putfield}\ \ f$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$A :: B :: X \triangleleft \hat{S}_\delta(m,pc) :$
$\forall(\mathsf{Ref}\ \sigma') \in B : A \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma')(f)$
$X \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$
$\hat{T}_\delta(m,pc) \subseteq \hat{T}_\delta(m,pc+1)$

$(\hat{H},\hat{L},\hat{S},\hat{T}) \models_{\text{TFA}} (m,pc) : \texttt{invokevirtual}\ \ m_0$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m,0) : \hat{T}_\delta(m,pc) \neq \emptyset \Rightarrow$
$A_1 :: \cdots :: A_{|m_0|} :: B :: X \triangleleft \hat{S}_\delta(m,pc) :$
$\forall(\mathsf{Ref}\ \sigma) \in B :$
$m_v = methodLookup(m_0, \sigma)$
$\forall \delta' \in \hat{T}_\delta(m,pc) :$
$\{\delta'\} \subseteq \hat{T}_{\delta'}(m_v, 0)$
$\{(\mathsf{Ref}\ \sigma)\} :: A_1 :: \cdots :: A_{|m_0|} \sqsubseteq \hat{L}_{\delta'}(m_v,0)[0..|m_0|]$
$\hat{T}_{\delta'}(m_v, \mathsf{END}) \subseteq \hat{T}_\delta(m,pc+1)$
$A :: Y \triangleleft \hat{S}_{\delta'}(m_v, \mathsf{END}) :$
$A :: X \sqsubseteq \hat{S}_\delta(m,pc+1)$
$\hat{L}_\delta(m,pc) \sqsubseteq \hat{L}_\delta(m,pc+1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{return}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\hat{T}_\delta(m, pc) \subseteq \hat{T}_\delta(m, \mathsf{END})$
$A :: X \triangleleft \hat{S}_\delta(m, pc) :$
$A :: \epsilon \sqsubseteq \hat{S}_\delta(m_0, \mathsf{END})$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.getTransactionDepth}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\hat{T}_\delta(m, pc) \sqsubseteq \hat{T}_\delta(m, pc + 1)$
$\forall \delta' \in \hat{T}_\delta(m, pc) : \{\delta'\} :: \hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.beginTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\{1\} \subseteq \hat{T}_\delta(m, pc + 1)$
$\hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.commitTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\{0\} \subseteq \hat{T}_\delta(m, pc + 1)$
$\hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$

$(\hat{H}, \hat{L}, \hat{S}, \hat{T}) \models_{\text{TFA}} (m, pc) : \texttt{API.abortTransaction}$
iff $\forall \delta \in \hat{T}_{\{0,1\}}(m, 0) : \hat{T}_\delta(m, pc) \neq \emptyset \Rightarrow$
$\{0\} \subseteq \hat{T}_\delta(m, pc + 1)$
$\hat{S}_\delta(m, pc) \sqsubseteq \hat{S}_\delta(m, pc + 1)$
$\hat{L}_\delta(m, pc) \sqsubseteq \hat{L}_\delta(m, pc + 1)$

# Bibliography


[ABF03] Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Notices*, 38(12):20–27, December 2003.

[Aga00a] Johan Agat. Transforming out Timing Leaks. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'00*, pages 40–53, Boston, Massachusetts, January 2000. ACM Press.

[Aga00b] Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology and Göteborg University, 2000.

[AJP95] Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society, 1995. Available online at `http://www.acsac.org/secshelf/book001/book001.html`.

[Ame02] Peter Amey. Correctness by Construction: Better Can Also Be Cheaper. *CrossTalk Magazine*, pages 24–28, March 2002.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[AR80] Gregory R. Andrews and Richard P. Reitman. An Axiomatic Approach to Information Flow in Programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison Wesley, 1985.

[Bai95] David Bailey. A Philosophy of Security Management. In Abrams et al. [AJP95], essay 3, pages 98–110. Available online at `http://www.acsac.org/secshelf/book001/book001.html`.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[BBD+03] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Automatic Validation of Protocol Narration. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*, pages 126–140, California, USA, June 2003. IEEE Computer Society.

[BBFG04] Frédéric Besson, Tomasz Blanc, Cédric Fournet, and Andrew D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *Proc. of the 17th IEEE Computer Security Foundations Workshop*, pages 61–75. IEEE Computer Society, June 2004.

[BBR04] Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security Types Preserving Compilation. In B. Steffen and G. Levi, editors, *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 2–15. Springer Verlag, 2004.

[BCC+] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technoogy Transfer (STTT)*. To appear. Preprint available at `ftp://ftp.cs.iastate.edu/pub/leavens/JML/sttt04.pdf`.

[BCG+00] Pierre Bieber, Jacques Cazin, Pierre Girard, Virginie Wiels Jean-Louis Lanet, and Guy Zanon. Checking Secure Interactions of Smart Card Applets. In *European Symposium on Research in Computer Security, ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 2000.

[BDNN98] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control Flow Analysis for the $\pi$-calculus. In D. Sangiorgi and R. de Simone, editors, *Proc. of conference on Concurrency Theory, CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98, Nice, France, September 1998. Springer Verlag.

[BDNN99] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static Analysis of Processes for No Read-Up and No Write-Down. In Wolfgang Thomas, editor, *Proc. of Foundations of Software Science and Computational Structures, FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 120–134, Amsterdam, The Netherlands, March 1999. Springer Verlag. Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'99).

[BDNN01a] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static Analysis for Secrecy and Non-Interference in Networks of Processes. In *Proc. of PACT'01*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer Verlag, 2001.

[BDNN01b] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static Analysis for the Pi-Calculus with Applications to Security. *Information and Computation*, 168:68–92, 2001.

[Bec01] Bernhard Beckert. A Dynamic Logic for the Formal Verification of Java Card Programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer Verlag, 2001.

[Ber97] Peter Bertelsen. Semantics of Java Byte Code. Student project report, Technical University of Denmark, 1997.

[Ber98] Peter Bertelsen. Dynamic semantics of Java byte-code. In *Workshop on Principles of Abstract Machines*, Pisa, Italy, September 1998. Proceedings published as a technical report of the computer science department of Universität des Saarlandes. Full version appears as [Ber97].

[BGH02] Gilles Barthe, Dilian Gurov, and Marieke Huisman. Compositional Verification of Secure Applet Interactions. In R.-D. Kutsche and H. Weber, editors, *Proc. of FASE'02*, volume 2306 of *Lecture Notes in Computer Science*, pages 15–32. Springer Verlag, 2002.

[Bis03] Matt Bishop. *Computer Security: Art and Science.* Addison Wesley, 2003.

[BL73a] David Elliot Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical Report ESD-TR-73-278, ESD/AFSC, Hanscom AFB, Bedford, Mass., November 1973. Also appears as MTR-2547, vol. 1, Mitre Corp., Bedford Mass. Digitally reconstructed in 1996.

[BL73b] David Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: A Mathematical Model. MITRE Technical Report 2547, Vol. 2, MITRE Corporation, May 1973. Digitally recontructed in 1996.

[BN02] Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop.* IEEE Computer Society, 2002.

[BNN02] Mikael Buchholtz, Hanne Riis Nielson, and Flemming Nielson. Experiments with Succinct Solvers. SECSAFE-IMM-002-1.0. Also published as DTU Technical Report IMM-TR-2002-4, February 2002.

[BS99] Gilles Barthe and Bernard P. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proc. of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer Verlag, 1999.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'77*, pages 238–252. ACM Press, 1977.

[CC99] Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation*, August 1999. Version 2.1. Also appears as International Standard ISO/IEC 15408:1999. Available for download at `http://www.commoncriteria.org`.

[CfP04] Centre for Pervasive Computing. Web page: `http://www.pervasive.dk`, January 2004.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison Wesley, 2000.

[CHS01] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context Inference for Static Analysis of Java Card Object Sharing. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security: Conference on Research in Smart Cards, E-smart 2001*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57, Cannes, France, September 2001. Springer Verlag.

[CJPR04] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In David Schmidt, editor, *Proc. of European Symposium on Programming, ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 385–400, Barcelona, Spain, March/April 2004. Springer Verlag.

[CKP03] Karl Crary, Aleksey Kliger, and Frank Pfenning. A Monadic Analysis of Information Flow Security with Mutable State. Technical Report CMU-CS-03-164, School of Computer Science, Carnegie Mellon University, July 2003.

[Coh77] Ellis S. Cohen. Information transmission in computational systems. In *Proc. of ACM Symposium on Operating Systems Principles, SOSP'77*, pages 133–139, West Lafayette, Indiana, USA, 1977. ACM Press.

[Coh78] Ellis S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, Devid P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computing*, pages 297–335. Academic Press, 1978.

[DD77] Dorothy Denning and Peter Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, 1976.

[DoD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, December 1985. DOD 5200.28-STD. Also known as "The Orange Book" and/or TCSEC. Obsoleted by [CC99].

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[ÉJ02] Marc Élouard and Thomas Jensen. Secure object flow analysis for Java Card. In *Proc. of Smart Card Research and Advanced Application Conference, Cardis'02*, 2002.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *New Security Paradigms Workshop 1999*, pages 87–95, Ontario, Canada, September 1999. ACM Press.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. Technical Report TR2000-1786, Department of Computer Science, Cornell University, 2000.

[FM98] Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'98*, pages 310–328, Vancouver, British Columbia, Canada, 1998. ACM Press.

[FM99] Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'99*, pages 147–166, Denver, CO, USA, November 1999. ACM Press.

[Fre98] Stephen N. Freund. Workshop on Formal Underpinnings of Java (FUJ'98). Available for download from `http://www-dse.doc.ic.ac.uk/~sue/oopsla/freund.f.ps`, October 1998.

[FSBJ97] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing Flexibility in Information Flow Control for Object-Oriented Systems. In *Proc. of the IEEE Symposium on Security and Privacy 1997*, pages 130–140, Oakland, CA, USA, May 1997. IEEE Computer Society.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *Proc. of the IEEE Symposium on Security and Privacy 2003*. IEEE Computer Society, May 2003.

[GM82] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy 1982*, pages 11–20, Oakland, California, USA, April 1982.

[GM84] Joseph A. Goguen and José Meseguer. Unwinding and Inference Control. In *Proc. of the IEEE Symposium on Security and Privacy 1984*, pages 75–86, 1984.

[GNN97] K.L.S. Gasser, F. Nielson, and H. Riis Nielson. Systematic realisation of control flow analyses for CML. In *Proc of ICFP'97*, pages 38–51. ACM Press, 1997.

[GS05] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, Paris, France, January 2005. Springer Verlag.

[Han02a] René Rydhof Hansen. A Prototype Tool for JavaCard Firewall Analysis. In *Proc. of the 7th Nordic Workshop on Secure IT-Systems, NordSec'02*, pages 35–53, November 2002. Proceedings published as Karlstad University Studies 2002:31.

[Han02b] René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0, 2002.

[Han02c] René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5, 2002.

[Han02d] René Rydhof Hansen. Implementing the Flow Logic for Carmel. SECSAFE-IMM-004-1.0, 2002.

[Han04] René Rydhof Hansen. A Hardest Attacker for Leaking References. In David Schmidt, editor, *Proc. of European Symposium*

*on Programming, ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 310–324, Barcelona, Spain, March/April 2004. Springer Verlag.

[HBL99] Pieter H. Hartel, Michael J. Butler, and Moshe Levy. The operational semantics of a Java Secure Processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 313–352. Springer Verlag, 1999.

[HGSC04] Marieke Huisman, Dilian Gurov, Christoph Sprenger, and Gennady Chugunov. Checking Absence of Illicit Applet Interactions: A Case Study. In *Proc of Formal Aspects of Software Engineering, FASE'04*, volume 2984 of *Lecture Notes in Computer Science*, pages 84–98, Barcelona, Spain, March/April 2004. Springer Verlag.

[HJNN99] René Rydhof Hansen, Jacob Grydholt Jensen, Flemming Nielson, and Hanne Riis Nielson. Abstract Interpretation of Mobile Ambients. In Agostino Cortesi and Gilbert Filé, editors, *Proc. Static Analysis Symposium, SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, pages 134–148, Venice, Italy, September 1999. Springer Verlag.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. The MIT Press, 2000.

[HM01] Pieter H. Hartel and Luc Moreau. Formalising the Safety of Java, the Java Virtual Machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.

[HMS03] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability Classes for Enforcement Mechanisms. Technical Report TR2003-1908, Computing and Information Science, Cornell University, August 2003.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[HS05] René Rydhof Hansen and Igor A. Siveroni. Towards Verification of Well-Formed Transactions in Java Card Bytecode. In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation, BYTECODE'05*, Electronic Notes in Theoretical Computer Science, Edinburgh, Scotland, April 2005. Elsevier. To appear.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java — A Minimal Core Calculus for Java and GJ. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'99*, Denver, CO, USA, November 1999. ACM Press.

[JML05] The Java Modeling Language (JML). Web page: `http://www.cs.iastate.edu/~leavens/JML/`, January 2005.

[key05] The key project. Web page: `http://i12www.ilkd.uni-karlsruhe.de/~key/`, January 2005.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology — CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397, Santa Barbara, California, USA, August 1999. Springer Verlag.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, Santa Barbara, California, USA, August 1996. Springer Verlag.

[Koz99] Dexter Kozen. Language-Based Security. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proc. Conf. Mathematical Foundations of Computer Science, MFCS'99*, volume 1672 of *Lecture Notes in Computer Science*, pages 284–298. Springer Verlag, 1999.

[KS02] Naoki Kobayashi and Keita Shirane. Type-Based Information Analysis for Low-Level Languages. In *Proc. of Asian Symposium on Programming Languages and Systems, APLAS'02*, pages 302–316, Shanghai, China, November/December 2002.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lan81] Carl E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition edition, 1999. Available for download at `http://java.sun.com/docs/books/vmspec/index.html`.

[Mar00] Renaud Marlet. Typical Code Patterns Found in Java Card Applets To Be Used as Targets for Program Analysis. SECSAFE-TL-003-1.0, December 2000.

[Mar01] Renaud Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project. SECSAFE-TL-005-1.7, May 2001.

[Mar02] Renaud Marlet. Demoney: Java Card Implementation. SECSAFE-TL-008-0.8, November 2002.

[McL85] John McLean. A Comment on the "Basic Security Theorem" of Bell and LaPadula. *Information Procesing Letters*, 20(2):67–70, February 1985.

[McL90] John McLean. The Specification and Modeling of Computer Security. *Computer*, 23(1), January 1990.

[McL94] John McLean. Security Models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[Mil81] Jonathan K. Millen. Information Flow analysis of Formal Specifications. In *Proc. of the IEEE Symposium on Security and Privacy 1981*, pages 3–8. IEEE Computer Society, 1981.

[Mil87] Jonathan K. Millen. Covert Channel Capacity. In *Proc. of the IEEE Symposium on Security and Privacy 1987*, pages 60–66. IEEE Computer Society, 1987.

[MM01] Renaud Marlet and Daniel Le Métayer. Security Properties and Java Card Specificities To Be Studied in the SecSafe Project. SECSAFE-TL-006-1.2, August 2001.

[MM02] Renaud Marlet and Cédric Mesnil. Demoney: A Demonstrative Electronic Purse (Card Specification). SECSAFE-TL-007-0.8, November 2002.

[MPH01] P. Müller and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Formal Techniques for Java Programs*, 2001.

[Nec97] George C. Necula. Proof-Carrying Code. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'97*, pages 106–119, Paris, France, January 1997. ACM, ACM Press.

[Neu95] Peter G. Neumann. *Computer Related Risks*. ACM Press/Addison Wesley, 1995.

[NHN03] Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Abstract Interpretation of Mobile Ambients. *Science of Computer Programming*, (47):145–175, 2003.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications – A Formal Introduction*. Wiley Press, 1992. Out of print. Revised edition (July 1999) available from `http://www.imm.dtu.dk/~riis/Wiley_book/wiley.html`.

[NN97] F. Nielson and H. Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'97*, pages 332–345. ACM Press, 1997.

[NN98a] F. Nielson and H. Riis Nielson. The flow logic of imperative objects. In *Proc. Conf. Mathematical Foundations of Computer Science, MFCS'98*, number 1450 in Lecture Notes in Computer Science, pages 220–228. Springer Verlag, 1998.

[NN98b] F. Nielson and H. Riis Nielson. Flow Logics and Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 10, 1998.

[NN98c] H. Riis Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. International Conference on Compiler Construction, CC'98*, number 1383 in Lecture Notes in Computer Science, pages 109–127. Springer Verlag, 1998.

[NN00] Hanne Riis Nielson and Flemming Nielson. Hardest Attackers. In *Workshop on Issues in the Theory of Security, WITS'00*, 2000.

[NN02] H. Riis Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Verlag, 2002.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[NNH02] Flemming Nielson, Hanne Riis Nielson, and René Rydhof Hansen. Validating Firewalls using Flow Logics. *Theoretical Computer Science*, 283(2):381–418, 2002.

[NNHJ99] Flemming Nielson, Hanne Riis Nielson, René R. Hansen, and Jacob G. Jensen. Validating Firewalls in Mobile Ambients. In *Proc. of conference on Concurrency Theory, CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 463–477. Springer Verlag, August 1999.

[NNS02a] F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *Electronic Notes in Theoretical Computer Science*, 62, 2002.

[NNS02b] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A Succinct Solver for ALFP. *Nordic Journal of Computing*, 2002(9):335–372, 2002.

[NNS+04] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The Succinct Solver Suite. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)04*, volume 2988 of *Lecture Notes in Computer Science*, pages 251–265, Barcelona, Spain, 2004. Springer Verlag.

[NS01] Flemming Nielson and Helmut Seidl. Succinct solvers. Technical Report 01-12, University of Trier, Germany, 2001.

[Oes99] Marcus Oestreicher. Transactions in Java Card. In *Proc. of the Annual Computer Security Applications Conference, ACSAC'99*, pages 291–298, Phoenix, Arizona, USA, December 1999. IEEE Computer Society.

[PBB+04] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing High-Level Security Properties for Applets. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A.A. El Kalam, editors, *Proc. of Smart Card Research and Advanced Application Conference, Cardis'04*, pages 1–16. Kluwer, 2004.

[PCC01] I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and Sharing Domains for Static Analysis of Java Programs. In *European Conference on Object-Oriented Programming, ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 77–98, Budapest, Hungary, June 2001. Springer Verlag.

[Pie02] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[Pil03] Henrik Pilegaard. A feasibility study. SECSAFE-IMM-008-1.0, September 2003.

[Plo81] Gordon D. Plotkin. A Structural Approach to Operational Semantics. DAIMI FN-19, Computer Science Department (DAIMI), Aarhus University, September 1981.

[RIS] The ACM Committee on Computers and Public Policy. *The RISKS Digest: The ACM Forum on Risks to the Public in Computers and Related Systems*. Moderated by Peter G. Neumann. Digests available from `http://www.risks.org`.

[RMMG01] Peter Ryan, John McLean, Jon Millen, and Virgil Gligor. Noninterference, who needs it? In *Proc of the 14th IEEE Computer Security Foundations Workshop*, pages 237–238, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society. Opening statement for panel discussion.

[RR98] Eva Rose and Kristoffer Høgsbro Rose. Lightweight Bytecode Verification. In *FUJ'98*, 1998. Extended abstract.

[Rus92] John Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, December 1992.

[SBCJ97] Pierangela Samarati, Elisa Bertino, Alessandro Ciampichetti, and Sushil Jajodia. Information Flow Control in Object-Oriented Systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July/August 1997.

[Sch00] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[SH01] Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2, October 2001.

[Sha48] Claude E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:329–423,623–656, July,October 1948. Reprinted with corrections.

[Siv03a] Igor Siveroni. Formalisation of the Semantics of Java Card. SECSAFE-ICSTM-014-2.0, October 2003.

[Siv03b] Igor Siveroni. SecSafe. Web page: `http://www.doc.ic.ac.uk/~siveroni/secsafe/`, 2003.

[Siv04] Igor Siveroni. Operational Semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, January–March 2004. Special issue on Formal Methods for Smart Cards.

[SJ03a] Igor Siveroni and Luke Jackson. Prototype Implementation of an Integrated Interpreter and Analyser of Carmel Programs. SECSAFE-ICSTM-015-1.0, October 2003.

[SJ03b] Fausto Spoto and Thomas Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.

[SJE01] Igor Siveroni, Thomas Jensen, and Marc Eluard. A Formal Specification of the Java Card Firewall. In Hanne Riis Nielson, editor, *Proc. of Nordic Workshop on Secure IT-Systems, NordSec'01*, pages 108–122, Lyngby, Denmark, November 2001. Proceedings published as DTU Technical Report IMM-TR-2001-14.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, January 2003.

[SMH01] Fred B. Schneider, Greg Morrisett, and Robert Harper. A Language-Based Approach to Security. In R.Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer Verlag, 2001.

[Ste98] Karen Stephenson. Towards an Algebraic Specification of the Java Virtual Machine. In B. Moeller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 236–277. Springer Verlag, 1998.

[Sun00] Sun Microsystems. *Java Card 2.1.1 Virtual Machine Specification*, May 2000.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984. Turing Award Lecture.

[TP00] Frank Tip and Jens Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'00*, pages 281–293, Minneapolis, MN, USA, October 2000. ACM Press.

[ver05] The VerifiCard Project. Web page: `http://www.verificard.org/`, January 2005.

[VHU92] Jan Vitek, R. Nigel Horspool, and James S. Uhl. Compile-Time Analysis of Object-Oriented Programs. In *Proc. International Conference on Compiler Construction, CC'92*, volume 641 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot — a Java Bytecode Optimization Framework. In *CASCON99*, September 1999.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java Bytecode using the Soot Framework: Is it Feasible? In *Proc. International Conference on Compiler Construction, CC'2000*, March/April 2000.

[VRH98] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report SABLE-TR-1998-4, McGill University, July 1998.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[WR99] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'99*, pages 187–206, Denver, CO, USA, November 1999. ACM Press.

[ZR02] Karen Zee and Martin Rinard. Write barrier removal by static analysis. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'02*, pages 191–210, Seattle, Washington, USA, 2002. ACM Press.