Static analysis of concurrent programs based on behavioral type systems

Abel Celestrín

Alma Mater Studiorum — Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXIX

Settore Concorsuale di Afferenza: 01/B1
Settore Scientifico Disciplinare: INF/01

Static analysis of concurrent programs based on behavioral type systems

Presentata da: Abel García Celestrín

Coordinatore Dottorato:
Paolo Ciaccia

Relatore:
Cosimo Laneve

Esame finale anno 2017

Data di discussione: 15 Maggio 2017

Abstract

The strength of program static analysis techniques lies on its ability to detect faulty behaviors prior to the execution. This ability requires that the analysis process foresees any possible runtime scenario. A task which is even more complex in the case of concurrent programs, because of the number of alternatives introduced by the usual nondeterminism. In this particular case, some of the most common faulty behaviors are those about erroneous usage of resources, presence of deadlocks and data race conflicts.

Behavioral type systems for programming languages provide a strong mechanism for reasoning on programs actions at static time. In this thesis we discuss two static analysis techniques based on this approach. The first one, targets the resource usage in an ad-hoc language with full-fledged operations for acquiring and releasing virtual machines. The second one, targets the deadlock analysis of Java programs.

In both cases we provide a formal proof of correctness, along with prototype implementations that allow practically to test the feasibility of these solutions. These prototypes have also allowed assessing these techniques against others existing in the literature obtaining very encouraging results.

Acknowledgements

During your life as Ph.D. Student several people crosses paths with you, in such a moment of your life you get to appreciate even the slightest act of support. To all of them I want to express my appreciation and gratefulness.

That said, there are some persons that in one way or another become special by living this experience by your side.

A very deep gratitude goes to my advisor Prof. Cosimo Laneve, I will always take with me the rewarding discussions, and the enormous support in the process of writing and describing scientific results. Of course, I will not forget either the passion for the science, and the stylish approach in presentations (which I have, secretly, tried to imitate a lot).

I also want to thank Elena Giachino for being not only a colleague but also a friend since day one, with whom I have shared amazing moments both professionally and personally speaking.

I want to thank my Ph.D. colleagues: Valeria, Stefano, Francesco, Vincenzo and Vincenzo "the master", and Giacomo, for contributing to create a really nice work environment, and for bringing a nice warm atmosphere to our cold basement. A special gratitude goes to Saverio Giallorenzo for being always two steps forward but showing the rest of us the right way ahead.

I feel very grateful as well to all the people in the Envisage Team. In particular, to Reiner Hanle who invited me to Germany and to his amazing team that make me feel at home there. And to Einar Broch and Elvira Albert who kindly accepted being part of the reviewing committee of this thesis.

Finally, I want to thank Yisleidy Linares (my Yiyi) who is my partner in life but also happens to be a Ph.D. colleague. I don't know how these three years would have resulted without having here supporting me every single day.

Contents

Abstract i

Acknowledgements ii

Contents iii

1 Introduction 1
- 1.1 Static analysis . . . 1
- 1.2 Behavioral type systems based analysis . . . 3
- 1.3 Problems tackled in this work . . . 5
- 1.4 Aim, objectives, and contribution of this work . . . 6
  - 1.4.1 Objectives . . . 6
  - 1.4.2 Contribution . . . 7
- 1.5 Document contents . . . 7

I The static resource analysis problem 9

2 Machine usage upper bounds in `vml` 11
- 2.1 Introduction . . . 11
- 2.2 Motivation and Problem Statement . . . 13
- 2.3 The language `vml` . . . 15
  - 2.3.1 Syntax . . . 15
  - 2.3.2 Semantics . . . 17
  - 2.3.3 Examples . . . 20
- 2.4 Determinacy of method's arguments release . . . 21
- 2.5 The behavioral type system of `vml` . . . 24
- 2.6 The analysis of `vml` behavioral types . . . 30
  - 2.6.1 Arguments' identities . . . 31
  - 2.6.2 (Re)computing argument's states . . . 32
  - 2.6.3 The translation function . . . 34

2.7 Correctness . . . 39
2.7.1 The extension of the typing to configurations . . . 40
2.7.2 Runtime types evolution . . . 41
2.7.3 The subject reduction theorem . . . 43
2.7.4 Correctness of the cost computation . . . 46
2.7.5 Final demonstration . . . 50
2.8 Related Work . . . 53
2.9 Conclusions . . . 55

3 The SRA tool 57
3.1 Introduction . . . 57
3.2 Type inference . . . 58
3.3 Translation of cost equations . . . 60
3.4 An illustrative example . . . 61
3.5 Related tools and assessments . . . 62
3.6 SRA Deliverable . . . 66
3.7 Conclusions . . . 67

II The deadlock analysis problem 69

4 Deadlock detection in JVML 71
4.1 Introduction . . . 71
4.2 Motivation and Problem Statement . . . 74
4.3 The language $\texttt{JVML}_d$ . . . 76
4.3.1 Syntax . . . 77
4.3.2 Informal semantics . . . 77
4.3.3 $\texttt{JVML}_d$ deadlock example: the `Network` class . . . 78
4.4 Preliminaries . . . 80
4.5 A behavioral type system for $\texttt{JVML}_d$ . . . 86
4.5.1 Typing rules . . . 86
4.5.2 The abstract behavior of the `Network` class . . . 89
4.6 Analysis . . . 91
4.7 `wait`, `notify` and `notifyAll` methods . . . 92
4.8 Type system extensions . . . 94
4.8.1 Static members . . . 94
4.8.2 Exceptions and exception hndlers . . . 96
4.8.3 Arrays . . . 98
4.8.4 Recursive data types . . . 101
4.8.5 Inheritance . . . 103
4.8.6 Alternative concurrent models and native methods . . . 105

4.9 Full proof of correctness . . . 106
4.9.1 Correctness overview . . . 106
4.9.2 The JVML formal semantics . . . 107
4.9.3 Lam semantics . . . 108
4.9.4 Flattening and circularities . . . 111
4.9.5 Later stage relation . . . 111
4.9.6 Runtime typing . . . 112
4.9.7 Subject Reduction . . . 113
4.9.8 Deadlocks and circularities . . . 120
4.10 Related work . . . 121
4.10.1 Static approaches . . . 121
4.10.2 Non-static approaches . . . 123
4.11 Conclusions . . . 124

5 The JaDA tool 127
5.1 Introduction . . . 127
5.2 Type inference . . . 128
5.3 Analysis of circularities . . . 132
5.4 Related tools and assessments . . . 133
5.5 JaDA Deliverable . . . 135
5.5.1 Customization . . . 137
5.6 Conclusions . . . 138

6 General conclusions 141
6.1 Conclusions . . . 141
6.2 Future work . . . 142

Bibliography 145

Chapter 1

Introduction

In this Chapter we present a general overview of the concepts and problems treated in this thesis. We start with a brief description of the purpose of program static analysis, and in particular, the behavioral type system based approach. Then, we concisely describe the two program verification problems that are solved by the static techniques presented in this work. Finally, we conclude with the main goals of this thesis and the structure of this document.

1.1 Static analysis

There are two big well-defined groups of approaches for program analysis: static analysis techniques and runtime analysis techniques. In addition, there are as well hybrid techniques that combine the previous two.

The main characteristic of static analysis techniques is that programs are verified without actually being executed. Usually such analyses are performed on the source code or, in other cases, on some form of the *object code* (the intermediate language generated by some compilers like `JVML` in the case of the `Java` virtual machine). The term *static analysis* is, in general, applied to a verification performed in an automatic way. There are human-aided static analysis tools as well, but these are usually referred in the literature as program understanding, program comprehension, or code review.

The vast majority of the existing programming languages use some sort of static analysis for detecting potential errors prior to the execution time. The simplest examples are perhaps strong-typed languages, that define a type system that is checked at compiling time for ensuring type safety.

However, other kind of properties (i.e. resource usage upper bounds, deadlock freedom, program termination and data race conflicts) may be much more difficult to verify.

The advantage of the static analysis tools is the possibility to reason about the whole set of possible program executions. Theoretically, such checking ensures a certain quality before the software deployment stage. In general, the static analysis is as complex as the properties targeted by the verification. Logically, when these properties become very complex, the main drawbacks of these approaches appear.

The main issues with static analysis techniques is that analyzing all possible program executions takes a considerable amount of time. It is important to consider that since static techniques take place during compilation time they are not as time sensitive as dynamic analysis techniques that take place during the program runtime. Still, even for not very big programs, analyzing the whole set of possible executions may lead to an exponential unmanageable problem. Therefore, for non-trivial programs, some level of abstraction is needed. Abstraction allows to scale the large set of possible program runs into a smaller (analyzable) model. Implicitly, this produces a common trade-off in static analysis techniques between precision and abstraction.

Formal methods. Program verification has been targeted by the Computer Science community since the very beginning. Perhaps one of the very first problems in this sense was Alan Turing's halting problem [57]. To this aim, several formal methods have been developed oriented to the analysis of computer programs. A static analysis technique may be purely based on one of these techniques, or on a combination of them. The following list briefly summarizes some of the most used formal methodologies:

Abstract interpretation: A technique based on the interpretation of an abstraction of the language semantics. This technique often uses over-approximations of the underlying program model in order to considerably reduce the set of possible program executions.

Model checking: This technique is built around the representation of a program by a finite set of states. These states are then analyzed separately and the resulting conclusions are combined to draw the overall verification of the program.

Symbolic execution: This technique tries to fill the gap between the static and dynamic analysis. The main idea is to group all the possible inputs of the program into sets that will produce similar behaviors. Afterwards, a symbolic representation of each set of inputs is evaluated by a symbolic interpreter. The final conclusions are drawn from the results of this symbolic execution.

Dataflow analysis: This is a fix-point based technique. Programs are represented by a control flow graph that corresponds to the source code. The states at each node of the program are then calculated repeatedly until a stable state is reached. Then, the verification is performed on each one of the control flow graph nodes.

1.2 Behavioral type systems based analysis

Behavioral type systems [11] provide a strong mechanism for reasoning on a program possible execution. With respect to standard type systems, the behavioral ones usually enrich standard types by adding some dynamic information of the effects of each expression.

Generic approach

The overall approach can be divided in four stages: (*i*) Parsing of the program, (*ii*) Typing process, (*iii*) Simplification of the program and (*iv*) Verification of the program.

The first step is usually performed after traditional static checks, thus the program is considered syntactically correct and with no obvious semantic errors. The main goal of this process is to obtain a representation of the program that is more suitable for the upcoming processes than the textual representation.

During the typing process the program behavioral type is obtained. This process can be fully automatic, partially automatic or manual. The typing process might not succeed, in that case the program could not be analyzed.

After a behavioral type is obtained, it is derived into an abstract simplified representation of the program. This abstraction drops features that are unrelated to the targeted analysis and focuses on those that are relevant. For example, in deadlock analysis, the abstraction may be an object dependency graph, for resource analysis it may be a set of cost equations.

Finally, the targeted properties are verified over the abstracted program. The verification process is only tied to the abstract program representation easing the use of third-party, and possibly more general, analyzers.

Advantages

The main advantage of this approach is its modularity. The type system is in general bounded to the language targeted by the analysis. The final verification is, in turn, tied to the abstract representation of the program.

This makes possible to apply the same analysis to different programming languages by having proper typing mechanisms producing the same kind of abstract representation.

This idea is illustrated in Figure 1.1. The figure corresponds to the deadlock analysis scheme for three different programming languages `Java`, `Scala` and `ABS`[1]. All three languages share the verification stage, since all three share the same simplified representation. However, there are two typing processes, one for `Java` and `Scala`, and the other for `ABS`. The first two languages can use the same inference mechanism since both of them are compiled to `Java` bytecode.

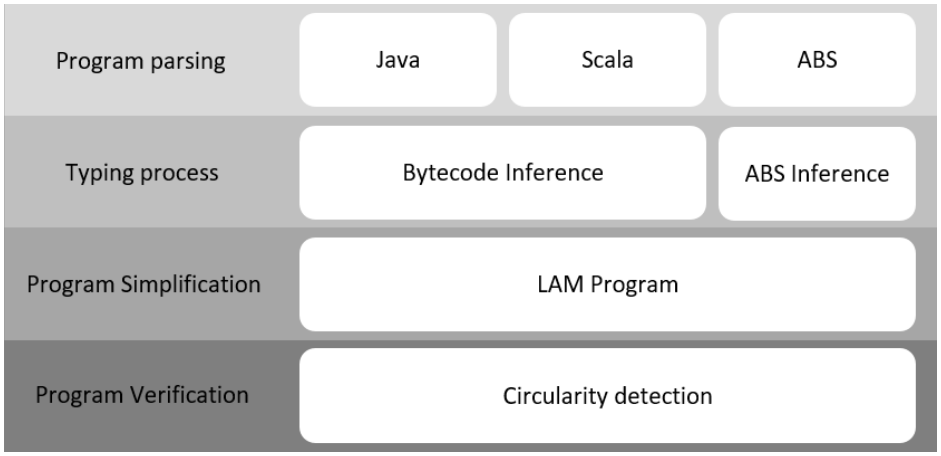

Figure 1.1: General approach of a behavioral type system based analysis

Another rather important advantage is that one can reason on the correctness of the analysis also in a modular way. For example, one can focus on two aspects separately: the typing and the abstraction, and the analysis. In the case of the typing process and its abstract representation, the correctness is usually proved by means of a subject reduction technique [24]. The correctness for the analysis will depend on the analysis technique. For example, in the case of the work described in Part I the correctness of the verification relies on the third party cost equation solver. In the case of the work described in Part II the analysis is based on a formally demonstrated decision algorithm.

[1] http://abs-models.org/

1.3 Problems tackled in this work

This work presents the theoretical basis and the prototype implementation of two static analysis tools based on behavioral type systems. The two problems targeted by these tools are the analysis of resource usage upper-bounds and the deadlock analysis. In both cases, the targeted language is an object-oriented language that allows concurrency operations.

The description of both tools is presented in the order they were conceived. This is an important remark, because the experience obtained in the development of the first tool has been key in the conception of the second one. In practice, it can be appreciated that the second of these tools presents a much deeper level of complexity. Also, in this second tool the result is more palpable since it can be applied to *real world* programming languages.

The analysis of virtual machines usage. In the first problem we analyze a domain specific language called `vml`. This is a rather simplistic language but expressive enough to allow the dynamic acquisition (creation) and releasing (deletion) of objects representing virtual machines. In this language, the operations on virtual machines can occur in parallel, and more over, virtual machine objects can run tasks that may, in turn, operate on other virtual machines. This scenario is inspired on the elastic features of the Cloud Computing environments. The purpose of the analysis of this language is to make an (over) estimation of the maximum number of virtual machines that may coexist during the program execution.

The resource analysis problem itself has received lots of attention from the scientific community. However, very few approaches target languages in which are both present, the concurrency feature and the explicit resource removal operation. Moreover, to the best of our knowledge, there are not previous works targeting the static analysis of code exploiting the Cloud elasticity features.

The analysis of deadlocks in `Java` bytecode. There have been several works targeting the analysis of deadlocks in the `Java` language and its byte code (`JVML`). The existing techniques, however, target either a very reduced subset of the language or take little to none consideration about the soundness of their approach. An interesting remark, is also the fact, that to the best of our knowledge none of the existing analyzers reports the analysis of Scala programs, a language that is also compiled into `JVML`.

1.4 Aim, objectives, and contribution of this work

The main goal of this thesis is to present the strength of the behavioral type system based approach for the static analysis in concurrent programs. To accomplish such purpose, we focus on two program verification problems of high relevance and with application in the newest developments in the Computer Science world.

It is also part of the aim of this work to produce competitive solutions to these problems at the level of other state of the art solutions. We also expect these solutions to be fully automatic and capable to perform with very little human interaction. This is a key feature for the purpose of analyzing non-trivial programs

1.4.1 Objectives

The main objective of this thesis is to provide valid tools for the analysis of the problems discussed in Section 1.3. Technically, the process of building such tools, with the approach followed in this work, involves the following partial objectives:

- **Definition of the behavioral type systems:** To design the static semantic rules that define the type systems for both of the problems tackled in this work.
- **Design and implementation of automatic behavioral type inference:** To implement a fully automatic mechanism for the inference of program types, according to the behavioral type systems designed and the targeted languages.
- **Proofs of correctness of the behavioral type systems:** To demonstrate the correctness of the behavioral type systems in characterizing the underlying programs and producing correct program abstractions without losing key information for the analysis.
- **Design and implementation of the analysis of the behavioral type systems:** To provide an automatic mechanism for drawing conclusions from the abstractions produced during the typing process.
- **Comparison of the result of the proposed tools with existing similar ones:** To demonstrate the effectiveness of the resulting tools by testing them against other related approaches.

1.4.2 Contribution

Two main contributions can be derived from the results of this work:

i the approach for the static analysis of resource utilization in a concurrent and object oriented language with explicit resource release operations.

ii the approach for the static analysis of deadlocks in `Java` bytecode.

Each of these solutions presents a custom and complex behavioral type system. The instrumentation of these type systems as well as the proof of correctness behind them are also a remarkable part of the results obtained in this work.

The design of these solutions cover issues that are typical in the static analysis of programs. The ways in which these issues are resolved constitute also relevant contributions of this work.

Finally, this work also provides a prototype implementation of the two solutions described. These prototypes constitute a simple way to interact with the results of these approaches making easy to compare it with previous and future equivalent techniques.

1.5 Document contents

The current document intends to provide an overall vision of the static analysis techniques based on behavioral type systems, and to present two successful applications of this approach. Following this introduction, the document is divided in two parts of two chapters each:

Part I: The static resource analysis problem

This part is dedicated to the analysis of resources in a concurrent object oriented language. The content is divided in two chapters.

Chapter 2: Machine usage upper bounds in `vml`, describes a variation of the static resource analysis problem applied to the Cloud Computing scenario. In particular, this problem focuses in the detection of upper bounds for the number of virtual machines necessary to execute a program in this environment. In this chapter, we describe a custom language: `vml`, which allows modeling the dynamic creation and release operations of the Cloud's elasticity feature. We then discuss a behavioral type system for abstracting virtual machine usage on these programs, and the subsequent

analysis of these types. This chapter concludes with the proof of correctness of the overall process.

Chapter 3: The `SRA` tool, describes a resource analysis tool based on the theory presented in Chapter 2. We discuss the main implementation details of this tool, specially the type inference process and the integration with an external cost equations solver. We present illustrative examples of the technique, and we also review the related literature and compare the tool and its technique with some of the existing approaches.

Part II: The deadlock analysis problem

This part is devoted to the deadlock analysis problem in Java programs. The content is divided in two chapters.

Chapter 4: Deadlock detection in JVML, describes the static deadlock analysis problem in `Java` programs. In this chapter, we present a behavioral type system for abstracting dependencies in `JVMLd`, a subset of the `Java` bytecode. We also present the algorithm for reasoning on the presence of circularities in the program dependencies. The chapter concludes with the extension of the behavioral type system to the full set of `Java` features and the proof of correctness of the overall approach.

Chapter 5: The `JaDA` tool, describes the `Java` deadlock analysis tool. We discuss the type inference process and the implementation of the circularities detection algorithm. In this chapter, we also review the related literature and compare the tool and its technique with some of the existing approaches including a commercial grade tool. We conclude with the description of the `JaDA` tool deliverables.

This document concludes with Chapter 6, where we present the main contributions of this thesis and describe some of the future possible lines of research that can be derived from it.

Part I

The static resource analysis problem

Chapter 2

Machine usage upper bounds in `vml`

Summary

We propose a *static analysis technique* that computes upper bounds of virtual machines usage in a *concurrent* language with explicit acquire and release operations over this kind of objects. In our language, the creation and release operations can occur in separated and possibly concurrent contexts (by passing virtual machine objects as arguments of invocations). Moreover, the objects representing virtual machines are active, that may be asynchronously hosting operations with side effects on the overall resource utilization. Our technique is modular and consists of (*i*) a type system associating programs with behavioral types that record relevant information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioral types and returns cost equations, and (*iii*) the integration with an automatic off-the-shelf solver for the cost equations.

2.1 Introduction

An accurate assessment of the resource usage in a computer program could reduce, for example, energy consumption and allocation costs. Two criteria that are even more important today, in modern architectures like mobile devices or Cloud Computing, where resources, such as virtual machines, have hourly or monthly rates.

While it is relatively easy to manually estimate worst-case costs for simple code examples, extrapolating this information for fully real-life complex programs could be cumbersome and highly error-sensitive. The first attempt

about the analysis of resource usage dates back to Wegbreit's pioneering work in 1975 [58], which develops a technique for deriving closed-form expressions out of programs. The evaluation of these expressions would return upper-bound costs that are parametrized by programs' inputs.

Wegbreit's contribution has two limitations: it addresses a simple functional language and it does not formalize the connection between the language and the closed-form expressions. A number of techniques have been developed afterwards to cope with more expressive languages (see [6, 20]) and to make the connection between programs and closed-form expressions precise (see [23, 41]). A more detailed discussion of the related work in the literature is presented in Section 2.8.

The existing cost analysis techniques vary also from the point of view of the resource targeted. The most common targets are probably the memory related resources like the heap space, and the number of steps in the program execution. However, some of the existing techniques also give certain abstraction on the type of the resource analyzed, by allowing code annotations with cost related expressions. Some attempts have even considered the analysis of the time necessary for the program execution [33].

This work targets the analysis of the usage of virtual machines represented by first order objects in a custom domain specific language called `vml`. The latter is a concurrent object-oriented language that includes explicit acquire and release operations on virtual machines object. These operations can occur in separated and possibly concurrent contexts (by passing virtual machine objects as arguments of invocations).

Our technique is modular and consists of (*i*) a type system associating programs with behavioral types that record relevant information for resource usage (creations, releases, and concurrent operations), (*ii*) a translation function that takes behavioral types and returns cost equations, and (*iii*) the integration with an automatic off-the-shelf solver for the cost equations.

To the best of our knowledge, current cost analysis techniques address (concurrent) languages where resource usage is always cumulative. We have found that only one of the existing techniques considers the scenario where both concurrent and resource removal operations are possible. That is the case of [8], however, our problem settings differ from these in two main aspects: i) the resources targeted by our technique are program objects that may carry, concurrently, operations that acquire and release other resources; and ii) in our case it is possible to have the acquire and release operations of a resource happening in different contexts by passing the objects as arguments of other possibly asynchronous method invocations.

Chapter contents

This Chapter starts by describing, in Section 2.2, the so called *Cloud Computing elasticity* concept that constitutes the inspiration of this work. In Section 2.3, we present `vml`: a simplified concurrent object-oriented language featuring virtual machine (VM) objects. In Section 2.4 we discuss some technical limitations that simplify, from the practical point of view, the analysis of `vml` programs. Section 2.5 and 2.6 present the definition of a behavioral type system for `vml` programs and its further analysis for estimating VM usage upper bounds. We then discuss, in Section 2.7, the correctness of this approach, supported by the formal proof of the soundness of the type system and the corresponding analysis. In Section 2.8 we review some of the main contributions in the literature related to the resource analysis problem. We conclude with the main remarks about this approach.

2.2 Motivation and Problem Statement

Cloud Computing introduces the concept of elasticity, whose main feature is the possibility for the resources, namely virtual machines (VM), to scale according to the software needs. There are two kinds of scaling, vertical and horizontal. Vertical Scaling means changing the deployed VM, adding more memory or disk space or using a more powerful processor, this kind of elasticity is, in general, associated to a disruptive workflow that stops the software execution and restarts it after the changes.

On the other hand, Horizontal Scaling is when the underlying number of VMs changes to fulfill the demand, this kind of scaling is a seamless process for the final user since it does not affect the availability of a service. Horizontal Scaling is the suggested scaling method by main Cloud providers like Amazon, Google and Microsoft Azure. This is appreciated in their pricing models that allow hiring, on demand, instances which are payed for the time they are in use; and also in their APIs that include instructions for requesting and releasing these VM instances on the fly.

In contrast to the elasticity concept, Cloud providers also give to the user the possibility to reserve a number of VM instances for a fixed period of time. This approach is used often by clients that migrate their existing solutions from private data centers to the Cloud. The main problem of this approach is the overcapacity or under-capacity of the allocated resources. The reserved VMs may spend important part of the time in an idle status, or the reserved VMs may be not enough to fulfill the demand. The table in Figure 2.1 shows some of the pricing possibilities given by the main Cloud providers. Notice

that the **on demand** price is a bit higher than the **reserved** alternative. However, the **on demand** model is more convenient when reserved machines may be idle more than the 40% of the time.

			On Demand		1-Year Reserved		
Class	Provider	Description	Hourly Rate	Monthly Rate	Hourly Rate	Monthly Rate	Saving
Micro	Azure	1 Core, 0.75 GB RAM	0.02	14.88			37.5%
	Amazon	1 Core, 1 GB RAM	0.013	9.672	0.009	6.696	30.8%
	Google	1 Core, 0.6 GB RAM	0.012	8.928	0.009	6.696	25.0%
Medium	Azure	1 Core, 3.5 GB RAM, 50 GB SSD	0.085	63.24			37.5%
	Amazon	1 Core, 3.75 GB RAM, 40 GB SSD	0.077	57.288	0.05	37.2	35.1%
	Google	1 Core, 3.57 GB RAM	0.063	46.872	0.045	33.48	28.6%
Large	Azure	16 Core, 112 GB RAM, 800 GB SSD	1.54	1145.76			37.5%
	Amazon	32 Core, 60 GB RAM, 640 GB SSD	1.68	1249.92	1.16	863.04	31.0%
	Google	16 Core, 104 GB RAM	1.184	880.896	0.82	610.08	30.7%

http://azure.microsoft.com/en-us/pricing/calculator/?scenario=virtual-machines
http://aws.amazon.com/ec2/purchasing-options/dedicated-instances/
https://cloud.google.com/pricing/

Figure 2.1: Some pricing options of main Cloud Providers, January 2016

Cloud computing elasticity APIs features acquire and release operations for the management of *On Demand* VMs (see for instance the *Amazon Elastic Compute Cloud* [50] or the *Docker Fiware*) that can be invoked dynamically from running code. The release operation happens to be a very powerful programming artifact, allowing to downscale a program's resource demand. It is worth noticing that, without a full release operation, the cost of a concurrent program may be modeled by aggregating the sets of operations that can occur in parallel, as in [7]. A full-fledged release operation, however, makes possible to delegate to other methods (possibly concurrent) the release of resources by passing them as arguments of invocations. For example, consider the following method

```
Int double_release(Vm x, Vm y) {
        release x;
        release y;
        return 0 ;
}
```

that takes two machines and simply releases them. The cost of this method will depend on the state of the machines in input:

- it may be `-2` when `x` and `y` are *different* and active;
- it may be `-1` when one machine is active and the other is released, or when `x` and `y` are *equal* and active – consider the invocation `double_release(x,x)`;

– it may be 0 when the two machines have been already released.

In this case, one might over-approximate the cost of `double_release` to 0. However, this leads to disregard releases and makes the analysis (too) imprecise. Moreover, consider the case where the machine objects can be hosting processes that either create or release other virtual machines. Ignoring a machine release may imply to assume the process hosted on it is still running, thus executing in parallel with the rest of the environment. This will have a serious impact in the calculation of the usage of these resources.

Problem statement

Lets us consider the following problem:

> Given a pool of on demand VM instances and a computer program that may indistinctly acquire and release these instances, what is the minimum cardinality of the pool that ensures the program runs without interruptions caused by the lack of available VMs? Assume that a VM can be re-acquired if and only if it has been previously released.

A solution to the previous problem has a direct application for both Cloud Providers and Cloud Costumers. For the former, it represents the possibility to know in advance how many real resources to allocate for an specific service. For the latter, it represents the possibility to pay *exactly* for the resources that are needed.

2.3 The language `vml`

The language `vml` is a future-based concurrent object-oriented language[1] with explicit acquire and release operations of virtual machines. In future-based languages, function/method invocations are *executed asynchronously* to the caller and are bound to variables called *futures*. The caller synchronizes with the callee by using the future variables when the return value is strictly needed. The syntax and the semantics of `vml` are defined in the following two subsections; the third subsection discusses a number of examples.

2.3.1 Syntax

In `vml` we distinguish between *simple types* T, which are either integers `Int` or virtual machines `VM`, and *future types* `Fut<`T`>`, which type asynchronous

[1] See [35] for more details on a similar language

invocations. The future argument T is instantiated with the return type of the invoked function. We use F to range over simple and future types. The notation $\overline{T\ x}$ denotes any finite sequence of *name declarations* $T\ x$ separated by commas. Similarly we write $\overline{T\ x\ ;}$ for a finite sequence of name declaration separated by semicolons.

A `vml` program is a sequence of method definitions $T\ \mathtt{m}(\overline{T\ x})\{\,\overline{F\ y\ ;}\ s\,\}$, ranged over M, plus a main body $\{\overline{F\,z\ ;}\ s'\}$. The syntax of statements s, expressions z and pure expressions e of `vml` is defined by the following grammar:

s	::=	$x = z$ | `if` $e\,\{\,s\,\}$ `else` $\{\,s\,\}$ | `return` e | $s\ ;\ s$ | `release` e	statements
z	::=	e | e`!m(`$\overline{e}$`)` | e`.get` | `new VM()`	expressions
e	::=	`this` | se | nse	pure expressions

A statement s may be either one of the standard operations of an imperative language plus the `release` e operation that disposes the virtual machine e.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method call $e!m(\overline{e})$ where e is the virtual machine that will execute the call, and $\overline{e}$ are the arguments of the call. This invocation does not suspend caller's execution: when the value computed by the invocation is needed then the caller performs a *non blocking* `get` operation, if the value needed by a process is not available then an awaiting process is scheduled and executed. Expressions z also include `new VM()` that creates a new virtual machine. The intended meaning of operations taking place on different virtual machines is that they may execute in parallel, while operations in the same virtual machine interleave their evaluation (even if in the following operational semantics the parallelism is not explicit).

A (*pure*) expression e may be the reserved identifier `this`, a virtual machines identifier or an integer expression. Since our analysis will be parametric with respect to the inputs, we parse integer expressions in a careful way. In particular we split them into *size expressions* se, which are expressions in Presburger arithmetics [18] (this is a decidable fragment of Peano arithmetics that only contains addition), and *non-size expressions* nse, which are the other type of expressions. The syntax of size and non-size expressions is the following:

nse	::=	k | x | $nse \leqslant nse$ | $\neg nse$ | nse `and` nse	non-size expressions
	|	nse `or` nse | $nse + nse$ | $nse - nse$	
	|	$nse \times nse$ | nse/nse	
se	::=	ve | $ve \leqslant ve$ | $\neg se$ | se `and` se | se `or` se	size expressions
ve	::=	k | x | $ve + ve$ | $k \times ve$	integer size expressions
k	::=	*integer constants* | *err*	

We will use the term $nse = nse'$ as an abbreviation of "$(nse \leqslant nse')$ `and` $(nse' \leqslant nse)$", and similarly for se. We notice that (non-size and size) expressions also contain the value err – see below the definition of $[\![e]\!]_l$ for the semantics of arithmetics expressions that contain err. In the whole document, we assume that sequences of declarations $\overline{T\ x}$ and method declarations $\overline{M}$ do not contain duplicate names. We also assume that `return` statements never have a continuation.

2.3.2 Semantics

The `vml` semantic is defined as a transition relation between *configurations*, noted cn and defined below

cn	::=	$\epsilon \mid \mathit{fut}(f,v) \mid \mathit{vm}(o,a,p,q) \mid \mathit{invoc}(o,f,\mathtt{m},\overline{v}) \mid cn\ cn$	configurations
p	::=	$\{l \mid \epsilon\} \mid \{l \mid s\}$	process
q	::=	$\epsilon \mid p \mid q\ q$	sets of processes
v	::=	*integer constants* $\mid o \mid f \mid \bot \mid \mathit{err}$	run-time values
a	::=	$\top \mid \bot$	machine states
l	::=	$[\cdots, x \mapsto v, \cdots]$	maps

Configurations are sets of elements – therefore, we identify configurations that are equal up-to associativity and commutativity – and are denoted by the juxtaposition of the elements $cn\ cn$; the empty configuration is denoted by ϵ. The transition relation uses two infinite sets of names: *VM names*, ranged over by o, o', $\cdots$ and *future names*, ranged over by f, f', $\cdots$. The function fresh() returns either a fresh VM name or a fresh future name; the context will disambiguate between the two. The elements of configurations are:

- *virtual machines* $\mathit{vm}(o,a,p,q)$ where o is a VM name; a is either $\top$ or $\bot$ depending on whether the machine is alive or dead; p is either $\{l \mid \epsilon\}$, representing a terminated statement, or $\{l \mid s\}$, representing an *active process*, where l maps each local variable to its value and s is the statement to execute; and q is the set of processes to evaluate.

- *future binders* $\mathit{fut}(f,v)$. When the value v is $\bot$ then the actual value of f has still to be computed.

- *method invocation messages* $\mathit{invoc}(o,f,\mathtt{m},\overline{v})$.

Runtime values v are either integers or virtual machines and future names, or $\bot$, meaning an un-computed value, or an erroneous value err. The fol-

lowing auxiliary functions are used in the semantic rules (we assume a fixed `vml` program):

- $\mathrm{dom}(l)$ returns the domain of l.
- $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$.
- $[\![e]\!]_l$ returns the value of e, possibly retrieving the values of the names that are stored in l. Regarding boolean operations, as usual, `false` is represented by `0` and `true` is represented by a value different from `0`. Arithmetic operations in `vml` are also defined on the value err: when one of the arguments is err, every arithmetic operation returns err. $[\![\overline{e}]\!]_l$ returns the tuple of values of $\overline{e}$. When e is a future name, the function $[\![\cdot]\!]_l$ is the identity. Namely $[\![f]\!]_l = f$. It is worth noticing that $[\![e]\!]_l$ is undefined whenever e contains a name that is not defined in l.
- $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}) = \{[\overline{x} \mapsto \overline{v},\ \mathrm{destiny} \mapsto f] \mid s[^{o}/_{\mathtt{this}}]\}$, where $T\ \mathtt{m}(\overline{T\ x})\{\overline{T'\ z}; s\}$ is a method of the program. We observe that, because of bind, the map l in processes $\{l \mid s\}$ also binds the special name destiny to a future value. We also observe that the local names $\overline{z}$ do not belong to $\mathrm{dom}([\overline{x} \mapsto \overline{v},\ \mathrm{destiny} \mapsto f])$.

The transition relation rules are collected in Figure 2.2. The rules are almost standard, except those about the management of virtual machines and method invocations, which we are going to discuss.

Rule (New-VM) creates a virtual machine and makes it alive. Rules (Release-VM) and (Release-VM-self) dispose a virtual machine by means of the operation `release` x: this amounts to update its state a to $\bot$. Once a virtual machine has been released, no operation can be performed anymore by it, except letting the processes on the queue returning err – see rules (Activate), (VM-Err-Return), and (Bind-Mtd-Err).

Rule (Async-Call) defines asynchronous method invocation $x = e\,!\mathtt{m}(\overline{e})$. This rule creates a fresh future name that is assigned to the identifier x. Rule (Bind-Mtd) applies an invocation message by adding to the alive callee virtual machine a process corresponding to the called method. In case the callee virtual machine is not live, either (Bind-Mtd-Err) or (Bind-Partial) is applied, which binds err to the future name. Rule (Read-Fut) allows the caller to retrieve the value returned by the callee. It is worth noticing that the semantic of `get` is different from that of `ABS` [46] or of [35] because it is not blocking with respect to other processes waiting to be executed on the

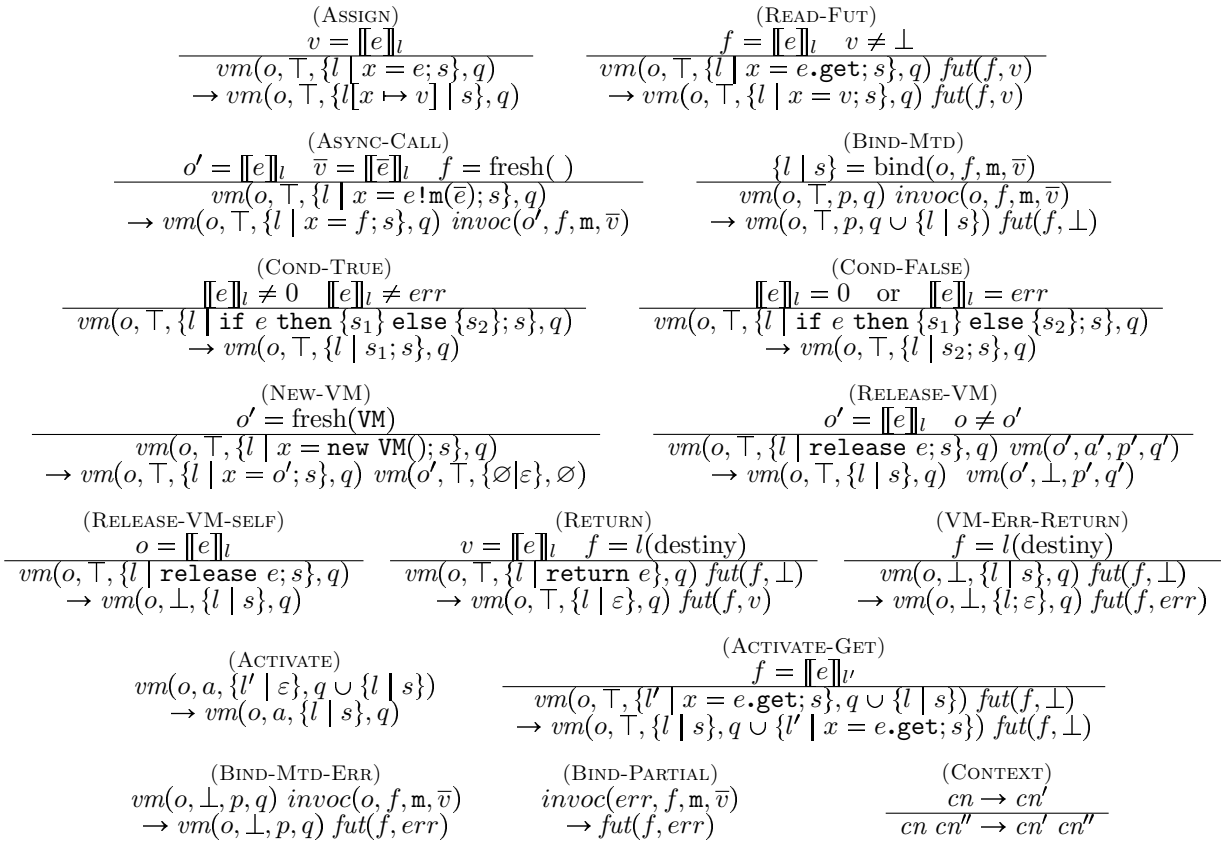

Figure 2.2: Semantics of `vml`.

same machine, see rule (Activate-Get). In fact, deadlock freedom is out of the scope of this contribution; in any case we refer to [35] for an algorithm (and a prototype) verifying deadlock freedom in `ABS`.

The initial configuration of a `vml` program with main body $\{\overline{F\ x\ ;}\ s\}$ is

$$vm(start, \top, \{[\text{destiny} \mapsto f_{start}] \mid s[^{start}/_{\texttt{this}}]\}, \varnothing)$$

where *start* is a special VM name and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$ and represents (part of) computations.

Problem statement reformulation

After discussing the semantics of `vml` we can now reformulate the Problem Statement from Section 2.2 in a more formal way. We start by defining the concept of *alive machines*:

Definition 3.1 (Alive machines)**.** *Given a configuration cn, a term* $vm(o, \top, p, q) \in cn$ *is called* alive machine *in cn. Let* $\texttt{alive}(cn)$ *be the number of different alive machines in cn.*

At this point we can rewrite the objective of this work as:

> to define a technique such that, given a configuration cn, it returns a n satisfying, for every $cn \longrightarrow^* cn'$, $\texttt{alive}(cn') \leqslant n$ (n is an upper bound to the alive machines in computations rooted at cn).

2.3.3 Examples

We now proceed to illustrate `vml` by discussing few examples and, for every example, we also examine the output we expect from our cost analysis. We begin with two methods computing the factorial function:

```
Int fact(Int n){
    Fut<Int> x ; Int m ;
    if (n==0) { return 1 ; }
    else { x = this!fact(n-1) ;
           m = x.get ;
           return m*n ;
    }
}
```

```
Int costly_fact(Int n){
    Fut<Int> x ; Int m ; VM z ;
    if (n==0) { return 1 ; }
    else { z = new VM();
       x = z!costly_fact(n-1) ; m = x.get ;
       release z; return m*n;
    }
}
```

The method `fact` is the standard definition of factorial with the recursive invocation `fact(n-1)` always performed on the same machine. That is, the computation of `fact(n)` only requires one virtual machine. On the contrary, the method `costly_fact` performs the recursive invocation on a new virtual machine `z`. The caller waits for its result, let it be `m`, then it releases `z` and delivers the value `m*n`. Since every virtual machine creation occurs before any release operation, `costly_fact` will create as many virtual machines as the argument n. That is, if the available resources are k virtual machines, then `costly_fact` can compute factorials up-to k.

The analysis of `costly_fact` has been easy because the `release` operation is applied to a locally created virtual machine. Yet, in `vml`, `release` can be also applied to method arguments and the presence of this feature in concurrent codes is a major source of difficulties for the analysis. A paradigmatic example is the `double_release` method discussed in Section 2.2 that may have either a cost of `-2` or of `-1` or of `0`.

It is worth observing that, while over-approximations (*e.g* not counting releases) return (too) imprecise costs, under-approximations may return

wrong costs. For example, the following method creates two virtual machines and releases the second one with `this!double_release(x,x)` before the recursive invocation.

```
Int fake_method(Int n) {
    if (n=0) return 0 ;
    else { VM x, y ; Fut<Int> f, g; Int u, v;
           x = new VM() ; y = new VM() ;
           f = this!double_release(x,x) ; u = f.get ;
           g = this!fake_method(n-1) ; v = g.get ;
           return 0 ; }
}
```

The cost of `fake_method(n)` should be `n`. However this is not the case if `double_release` is under-approximated with cost `-2`: In such case, one would wrongly derive a cost `0` of `fake_method(n)`. In Section 3.4 we consider an erroneous `fake_method` that increases the argument of the recursive invocation (instead of decreasing it) and discuss the corresponding cost equations returned by our technique. We notice that, in this case, the amount of virtual machines used by the erroneous method is infinite. The aim of the following sections is to present a technique for determining the cost of method invocations. Such a technique has to be sensible to the identity, and to the state of method's arguments before and after the invocation.

2.4 Determinacy of method's arguments release

Our cost analysis of virtual machines uses abstract descriptions that carry informations about concurrent method invocations and about creations and removals of virtual machines. In order to ease the compositional reasonings, method's abstract descriptions also define the arguments the method releases upon termination, called *method's effects*. In this contribution we stick to method descriptions that are as simple as possible, namely we assume that method's effects are *sets*. In turn, this requires methods' behaviours to be *deterministic* with respect to method's effects and, to enforce this determinacy, we define the following simplification properties.

The following simplifications allow us to focus on the relevant problems of the resource analysis of `vml`. It is possible to drop most of the simplifications by extending the approach presented in this work using almost standard solutions (for instance, with guarded or nondeterministic effects, see below). However these solutions would entangle a lot the technicalities of this presentation, making more difficult its reading and comprehension.

Simplification 1: *the branches in a method body always release the same set of method's arguments.* For example, methods like

```
Int foo1(VM x, Int n) {
    if (n = 0) return 0 ;
    else { release x ; return 0; }
}
```

does not follow this simplification because the then-branch does not release anything while the else-branch releases the argument `x`.

To analyze such program, one would just have to extend the presented approach with *guarded effects*, i.e., mapping from execution branches to effects, which can precisely identify the effects of each execution branch of the method.

Simplification 2: *method invocations are always synchronized within caller's body.* This implies that method's effects occur upon method termination. For example, in

```
Int foo2(VM x, VM y) {
    this!double_release(x,y) ; return 0 ;
}
```

the method's effects of `foo2` is not deteministic because `double_release` might still be running *after* `foo2` termination. This means that the termination of `foo2` has no impact on the termination of `double_release`. Therefore, the effect of `foo2` is empty – the caller cannot assume that `x` and `y` have been released upon synchronizing with `foo2`. Overall, this simplification supports a more precise analysis. In order to drop this restriction is necessary to add the unsynchronized invocations to the effects of the method. This idea is explained in [35] and also in Section 4.6 The underlying intuition is to pass the effects of unsynchronized invocations to the context, and to consider context effects along with the local effects of each method during the analysis phase.

Simplification 3: *machines executing methods with nonempty effects must be alive.* (This includes the carrier machine, *e.g.* method bodies cannot release the `this` machine.) At static time "alive" means that the machine is either the caller or has been locally created and has not been/is not being released. For example, in `foo3`

```
Int simple_release(VM x) {
     release x; return 0;
}
Int foo3(VM x) {
     VM z ; Fut<Int> f ; Int u ;
     z = new VM();
     f = z!simple_release(x);
     release z; u = f.get; return 0;
}
```

```
Int foo4(VM x, VM y) {
     Fut<Int> f ; Int u ;
     f = x!simple_release(y) ;
     u = f.get ; return 0 ;
}
```

the machine `z` is released before the synchronization with the `simple_release` statement `f.get`. This means that the disposal of `x` depends on the scheduler's choice and, in turn, it is not possible to determine whether `foo3` will release `x` or not. A similar non-determinism arises when the callee of a method releasing arguments is itself an argument. For example, in the above `foo4` method, it is not possible to determine whether `y` is released or not because we have no clue about `x`, being it an argument of `foo4`.

To analyze such programs, one would simply need to extend the approach presented in this work with non-deterministic effects.

Simplification 4: *if a method returns a machine, the machine must be new.* For example, consider the following code:

```
VM identity(VM x) { return x; }

{
   VM x ; VM y ; VM z ; Fut<VM> f ; Fut<Int> g ; Int m ;
   x = new VM() ; y = new VM() ;
   f = y!identity(x) ;
   g = this!simple_release(y);
   z = f.get ; m = g.get ;
   release z ;
}
```

In this case it is not possible to determine whether the value of `z` is `x` or `err` and, therefore, it is not clear whether the cost of `release z` is `0` or `-1`. The non-determinism is caused by `identity`, which returns the argument that is going to be released by a parallel method.

To analyze programs with such methods, one would simply need to extend the approach presented in this work with non-deterministic effects.

Simplifications 1, 3, and 4 are enforced by the type system in Section 2.5, in particular simplification 1 by rule (T-Method), simplification 3 by rules

$$
\begin{array}{rcll}
\mathfrak{O} & ::= & \_ \mid \alpha & \text{basic value} \\
\mathfrak{t} & ::= & \alpha \mid \alpha\downarrow \mid \partial \mid \bot \mid \top & \text{vm value} \\
\mathtt{op}' & ::= & + \mid - \mid = \mid \leqslant \mid \geqslant \mid \wedge \mid \vee & \text{linear operation} \\
\mathbb{r}, \mathbb{s} & ::= & \mathfrak{O} \mid se & \text{typing value} \\
\mathbb{z} & ::= & (\mathfrak{O}, \alpha, \mathtt{F}\mathfrak{t}, \mathtt{R}) \mid \mathfrak{O} & \text{future value} \\
\mathbb{x} & ::= & \_ \mid \mathtt{F}\mathfrak{t} \mid f \mid \mathbb{z} & \text{extended value} \\
\mathfrak{a} & ::= & \mathtt{0} \mid \nu\alpha \mid \nu f : \mathtt{m}\ \alpha(\overline{\mathbb{s}}) \mid \alpha^{\checkmark} \mid f_{\mathfrak{O}}{}^{\checkmark} & \text{atom} \\
\mathfrak{c} & ::= & \mathfrak{a} \rhd \Gamma \mid \mathfrak{a} \,;\, \mathfrak{c} \mid (se)\{\mathfrak{c}\} \mid \mathfrak{c} + \mathfrak{c} & \text{behavioral type}
\end{array}
$$

Figure 2.3: Behavioral Types Syntax

(T-Invoke) and (T-Release), and simplification 4 by rules (T-Invoke) and (T-Return).

2.5 The behavioral type system of vml

In this context, *Behavioral types* are abstract codes highlighting the features of vml programs that are relevant for the resource cost analysis in Section 2.6. These types support compositional reasonings and are associated to programs by means of a set of type system rules that are defined in this section.

The syntax of behavioral types uses *vm names* α, β, γ, $\cdots$, and *future names* f, f', $\cdots$. Sets of VM names will be ranged over by $\mathtt{S}$, $\mathtt{S}'$, $\mathtt{R}$, $\cdots$, and sets of future names will be ranged over by $\mathtt{F}$, $\mathtt{F}'$, $\cdots$. We assume that *err* is a *special* VM name representing the erroneous machine; therefore VM names will also range over *err*. The syntactic rules of these types are presented in Figure 2.3.

Behavioral types $\nu\alpha$ and $\alpha^{\checkmark}$ express creations of virtual machines and their removal, respectively. The type $\nu f : \mathtt{m}\ \alpha(\overline{\mathbb{s}})$ defines method invocations and $f_{\mathfrak{O}}{}^{\checkmark}$ defines the corresponding synchronization with the computation of the future f, in this case $\mathfrak{O}$ may be either α, which acts like a binder to the name of the returned virtual machine, or $\_$ if the target of f returns a different type. The conditional type $(se)\{\mathfrak{c}\}$ behaves like $\mathfrak{c}$ whenever se is true; the type $\mathfrak{c} + \mathfrak{c}'$ models nondeterminism. We will always shorten the type $f_{\_}{}^{\checkmark}$ into $f^{\checkmark}$.

In order to have a more precise type of continuations, the leaves of behavioural types are labelled with *environments*, ranged over by Γ, Γ', $\cdots$. Environments are maps from method names $\mathtt{m}$ to terms $\alpha(\overline{\mathbb{r}}) : \mathfrak{O}, \mathtt{R}$, from names to extended values $\mathbb{x}$, from future names to future values, and from VM names to extended values $\mathtt{F}\mathfrak{t}$, which are called *vm states* in the following. We assume that $\Gamma(err) = \varnothing\bot$. These environments occurring in the leaves

$$\text{(T-Var)}\ \frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-Primitive)}\ \Gamma \vdash k : k \qquad \text{(T-Op)}\ \frac{\Gamma \vdash e_1 : se_1 \qquad \Gamma \vdash e_2 : se_2}{\Gamma \vdash e_1\ \mathtt{op}'\ e_2 : se_1\ \mathtt{op}'\ se_2} \qquad \text{T-Unit}\ \frac{\Gamma \vdash e : se}{\Gamma \vdash e : \_}$$

$$\text{(T-Op-Unit)}\ \frac{\Gamma \vdash e_1 : \_ \quad \mathtt{or} \quad \Gamma \vdash e_2 : \_ \quad \mathtt{or} \quad \mathtt{op} \in \{*, /\}}{\Gamma \vdash e_1\ \mathtt{op}\ e_2 : \_} \qquad \text{(T-Pure)}\ \frac{\Gamma \vdash e : \mathbb{x}}{\Gamma \vdash e : \mathbb{x}, \mathtt{0} \rhd \Gamma}$$

$$\text{(T-Method-Sig)}\ \frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R} \quad \overline{\beta} \subseteq fv(\alpha, \overline{\mathbb{r}}, \mathbb{o}) \\ \sigma \text{ is a VM renaming such that } \mathbb{o} \notin fv(\alpha, \overline{\mathbb{r}}) \text{ implies } \sigma(\mathbb{o}) \text{ fresh}\end{array}}{\Gamma \vdash \mathtt{m}\ \sigma(\alpha)(\sigma(\overline{\mathbb{r}})) : \sigma(\mathbb{o}), \sigma(\mathtt{R})}$$

Figure 2.4: Typing rules for pure expressions

are only used in the typing proofs and are dropped in the final types (method types and the main statement type).

VM states $\mathtt{F}\mathbb{t}$ are a collection $\mathtt{F}$ of future names plus the value $\mathbb{t}$ of the virtual machines. This $\mathtt{F}$ specifies the set of parallel methods that are going to release the virtual machine; $\mathbb{t}$ defines whether the virtual machine is alive ($\top$), or it has been already released ($\bot$) or, according to scheduler's choices, it may be either alive or released (∂). VM values also include terms α and $\alpha\!\downarrow$. The value α is given to the argument machines of methods (they will be instantiated by the invocations – see the cost analysis in Section 2.6), the value $\alpha\!\downarrow$ is given to argument values that are returned by methods *and* can be released by parallel methods ($\alpha\!\downarrow$ will be also evaluated in the cost analysis). VM values are partially ordered by the relation $\leqslant$ defined by

$$\partial \leqslant \top \qquad \partial \leqslant \bot \qquad \alpha\!\downarrow \leqslant \bot \qquad \alpha\!\downarrow \leqslant \alpha\ .$$

In the following we will use the partial operation $\mathbb{t} \sqcap \mathbb{t}'$ returning, whenever it exists, the greatest lower bound between $\mathbb{t}$ and $\mathbb{t}'$. For example $\top \sqcap \bot = \partial$, but $\partial \sqcap \alpha\!\downarrow$ is not defined.

The type system uses judgments of the following form:

- $\Gamma \vdash e : \mathbb{x}$ for pure expressions e, $\Gamma \vdash f : \mathbb{z}$ for future names f, and $\Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{r}}) : \mathbb{o}, \mathtt{R}$ for methods.

- $\Gamma \vdash_{\mathtt{S}} z : \mathbb{x}, \mathbb{c} \rhd \Gamma'$ for expressions z, where $\mathbb{x}$ is the value, $\mathbb{c}$ is the behavioral type for z and Γ' is the environment Γ *with updates* of names and future names.

– $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c}$, in this case the updated environments are inside the behavioural type Γ', in correspondence of every branch of its.

The index $\mathtt{S}$ in the judgments for expressions and statements defines the set of method's arguments – see rule (T-Method) – and is used in the rule (T-Return) in order to constrain that the returned machine, if any, does not belong to method's arguments (*cf.* Restriction 4 in Section 2.4).

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$ and the environment update

$$\Gamma[x \mapsto \mathbb{x}](y) \stackrel{def}{=} \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

With an abuse of notation (see rule (T-Return)), we let $\Gamma[\_ \mapsto \mathbb{x}] \stackrel{def}{=} \Gamma$ (because $\_$ does not belong to any environment).

We will also use the operation and notation below:

– $\mathtt{F}\mathbb{t}\Downarrow$ is defined as follows:

$$\mathtt{F}\mathbb{t}\Downarrow \stackrel{def}{=} \begin{cases} \mathbb{t} & \text{if } \mathtt{F} = \varnothing \\ \partial & \text{if } \mathtt{F} \neq \varnothing \text{ and } \mathbb{t} = \top \\ \alpha\downarrow & \text{if } \mathtt{F} \neq \varnothing \text{ and } \mathbb{t} = \alpha \end{cases}$$

and, in Section 2.6, we write $(\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n)\Downarrow$ for $(\mathtt{F}_1\mathbb{t}_1\Downarrow, \cdots, \mathtt{F}_n\mathbb{t}_n\Downarrow)$.

– the *multihole contexts* $\mathcal{C}[\,]$ defined by the following syntax:

$$\mathcal{C}[\,] ::= \quad [\,] \quad | \quad \mathbb{a} \fatsemi \mathcal{C}[\,] \quad | \quad \mathcal{C}[\,] + \mathcal{C}[\,] \quad | \quad (se)\{\mathcal{C}[\,]\}$$

and, whenever $\mathbb{c} = \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n]$, then $\mathbb{c}[x \mapsto \mathbb{x}]$ is defined as $\mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1[x \mapsto \mathbb{x}]] \cdots [\mathbb{a}_n \rhd \Gamma_n[x \mapsto \mathbb{x}]]$.

The type system for expressions is reported in Figure 2.4. It is worth noticing that this type system is not standard because (size) expressions containing method's arguments are typed with the expressions themselves. This is crucial in the cost analysis of Section 2.6. It is also worth observing that, by rule (T-Primitive), $\Gamma \vdash \mathit{err} : \mathit{err}$ (while $\Gamma(\mathit{err}) = \varnothing\bot$). This expedient allows us to save one rule when typing method invocations with either erroneous or released carriers (*cf.* rule (T-Invoke-Bot)).

The type system for expressions with side effects and statements is reported in Figure 2.5. We discuss rules (T-Invoke), (T-Get), (T-Release), and (T-Return). Rule (T-Invoke) types method invocations $e!\mathtt{m}(\overline{e})$ by using a fresh future names. In particular, in the behavioral type $\nu f : \mathtt{m}\ \alpha(\overline{\mathbb{s}}) \to \beta$,

the (fresh) future name f is associated to the the method, the VM name of the callee, the arguments *and to the returned value*. This last value is fundamental when the invocation returns a new machine because, in this case, the type acts as a binder of β. Another important remark is about the value of f in the updated environment. This value contains the returned value, the VM name of the callee and its state, and the set of the arguments that the method is going to remove. The VM state of the callee will be used when the method is synchronized to update the state of the returned object, if any (see rule (T-GET)). It is also important to observe that the environment returned by (T-INVOKE) is updated with information about VM names released by the method: every such name will contain f in its state. Let us now discuss the constraints in the second and third lines of the premise of (T-INVOKE). As regards the second line, assuming that the callee has not been already released ($\Gamma(\alpha) \neq \mathtt{F}\bot$), there are two cases:

(i) either $\Gamma(\alpha) = \varnothing\top$ or α is the caller object α': namely the callee is alive because it has been created by the caller or it is the caller itself,

(ii) or $\Gamma(\alpha) \neq \varnothing\top$: this case has two subcases, namely either ($ii.a$) the callee is being released by a parallel method or ($ii.b$) it is an argument of the caller method – see rule (T-METHOD).

While in (i) we admit that the invoked method releases VM names, in case (ii) we forbid any release, as we discussed in Restriction 3 in Section 2.4.

We observe that, in case ($ii.b$), being α an argument of the method, it may retain any state when the method is invoked and, for reasons similar to ($ii.a$), it is not possible to determine at static time the exact subset of $\mathtt{R}$ that will be released. The constraint in the third line of the premise of (T-INVOKE) enforces Restriction 3 to the other invocations in parallel and to the object executing $e!\mathtt{m}(\overline{e})$.

Rule (T-GET) defines the synchronisation with a method invocation that corresponds to a future f. Let $(\mathtt{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})$ be the value of f in the environment. There are two cases: either (i) $\mathtt{R} \neq \varnothing$ or (ii) $\mathtt{R} = \varnothing$. In case (i), by Restriction 3, $\mathtt{F}\mathbb{t}$, which is the value of the caller α when the method has been invoked, *is equal* to the value $\Gamma(\alpha)$ (and $\mathtt{F} = \varnothing$). In this case, if the method returns a new machine (*cf.* Restriction 4), its state must be $\varnothing\mathbb{t}$ (notice that $\mathbb{t} \sqcap \Gamma(\alpha) \Downarrow = \mathbb{t}$). The case ($ii$) is more problematic because the caller may be released by a method running in parallel and, therefore, the possible returned virtual machine must record this information in its state.

$$\text{(T-Assign-Var)}\quad \frac{\Gamma(x) = \mathbb{x} \qquad \Gamma \vdash_{\mathtt{S}} z : \mathbb{x}', \mathbb{c}}{\Gamma \vdash_{\mathtt{S}} x = z : \mathbb{c}[x \mapsto \mathbb{x}']}$$

$$\text{(T-Invoke)}\quad \frac{\begin{array}{c}\Gamma \vdash e : \alpha \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{s}} \qquad \Gamma \vdash \mathtt{m}\,\alpha(\overline{\mathbb{s}}) : \mathbb{o}, \mathtt{R} \qquad \Gamma \vdash \mathtt{this} : \alpha' \qquad \Gamma(\alpha) \neq \mathtt{F}\bot \\ (\Gamma(\alpha) \neq \varnothing\top \quad \text{and} \quad \alpha \neq \alpha') \qquad \text{implies} \qquad \mathtt{R} = \varnothing \\ \mathtt{R} \cap \Big(\{\alpha'\} \cup \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \mathbb{x}, \mathtt{R}') \text{ and } \mathtt{R}' \neq \varnothing\}\Big) = \varnothing \\ f \text{ fresh} \qquad \Gamma' = \Gamma[\beta \mapsto (\{f\} \cup \mathtt{F}')\mathbb{t}]^{\beta \in \mathtt{R}, \Gamma(\beta) = \mathtt{F}'\mathbb{t}}\end{array}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \nu f\colon \mathtt{m}\ \alpha(\overline{\mathbb{s}}) \rhd \Gamma'[f \mapsto (\mathbb{o}, \alpha, \Gamma(\alpha), \mathtt{R})]}$$

$$\text{(T-Invoke-Bot)}\quad \frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash \overline{e} : \overline{\mathbb{s}} \qquad \Gamma(\alpha) = \mathtt{F}\bot \qquad f \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} e!\mathtt{m}(\overline{e}) : f, \nu f\colon \mathtt{m}\ \alpha(\overline{\mathbb{s}}) \rhd \Gamma[f \mapsto (err, \alpha, \mathtt{F}\bot, \varnothing)]}$$

$$\text{(T-Get)}\quad \frac{\begin{array}{c}\Gamma \vdash x : f \qquad \Gamma \vdash f : (\mathbb{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R}) \\ \mathtt{R}' = fv(\mathbb{o}) \backslash \mathtt{R} \qquad \mathbb{t}' = \mathbb{t} \sqcap (\Gamma(\alpha)\Downarrow) \\ \Gamma' = \Gamma[\beta \mapsto \varnothing\bot]^{\beta \in \mathtt{R}}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta' \in \mathtt{R}'}\end{array}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, f_{\mathbb{o}}{}^{\checkmark} \rhd \Gamma'[f \mapsto \mathbb{o}]}$$

$$\text{(T-Get-Done)}\quad \frac{\Gamma \vdash x : f \qquad \Gamma \vdash f : \mathbb{o}}{\Gamma \vdash_{\mathtt{S}} x.\mathtt{get} : \mathbb{o}, \mathtt{0} \rhd \Gamma}$$

$$\text{(T-New)}\quad \frac{\beta \text{ fresh}}{\Gamma \vdash_{\mathtt{S}} \mathtt{new\ VM}() : \beta, \nu\beta \rhd \Gamma[\beta \mapsto \varnothing\top]}$$

$$\text{(T-Release)}\quad \frac{\begin{array}{c}\Gamma \vdash x : \alpha \qquad \Gamma(\alpha) \neq \mathtt{F}\bot \\ \alpha \notin \{\beta \mid f' \in \mathrm{dom}(\Gamma) \text{ and } \Gamma(f') = (\mathbb{o}', \beta, \mathbb{x}, \mathtt{R}') \text{ and } \mathtt{R}' \neq \varnothing\}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}\ x : \alpha^{\checkmark} \rhd \Gamma[\alpha \mapsto \varnothing\bot]}$$

$$\text{(T-Release-Bot)}\quad \frac{\Gamma \vdash x : \alpha \qquad \Gamma(\alpha) = \mathtt{F}\bot}{\Gamma \vdash_{\mathtt{S}} \mathtt{release}\ x : \mathtt{0} \rhd \Gamma}$$

$$\text{(T-If)}\quad \frac{\Gamma \vdash e : se \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : (se)\{\, \mathbb{c}_1 \,\} + (\neg se)\{\, \mathbb{c}_2 \,\}}$$

$$\text{(T-If-ND)}\quad \frac{\Gamma \vdash e : \_ \qquad \Gamma \vdash_{\mathtt{S}} s_1 : \mathbb{c}_1 \qquad \Gamma \vdash_{\mathtt{S}} s_2 : \mathbb{c}_2}{\Gamma \vdash_{\mathtt{S}} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2}$$

$$\text{(T-Seq)}\quad \frac{\Gamma \vdash_{\mathtt{S}} s_1 : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \qquad (\Gamma_i \vdash_{\mathtt{S}} s_2 : \mathbb{c}'_i)^{i \in 1..n}}{\Gamma \vdash_{\mathtt{S}} s_1; s_2 : \mathcal{C}[\mathbb{a}_1 \fatsemi \mathbb{c}'_1] \cdots [\mathbb{a}_n \fatsemi \mathbb{c}'_n]}$$

$$\text{(T-Return)}\quad \frac{\begin{array}{c}\Gamma \vdash e : \mathbb{o} \qquad \Gamma \vdash \mathtt{destiny} : \mathbb{o}' \\ \mathbb{o} \notin \mathtt{S}\end{array}}{\Gamma \vdash_{\mathtt{S}} \mathtt{return}\ e : \mathtt{0} \rhd \Gamma[\mathbb{o}' \mapsto \Gamma(\mathbb{o})]}$$

Figure 2.5: Type rules for expressions and statements.

Let $\mathtt{F}\mathbb{t}$ be the state of the caller (which is recorded in $\Gamma(f)$). We use the operation $\mathbb{t} \sqcap \Gamma(\alpha) \Downarrow$ to this purpose, which means that the returned machine gets the same state of the carrier α if no method is releasing α, otherwise its state is either $\varnothing\partial$ or $\varnothing\alpha \downarrow$, according to the state of α was $\mathtt{F}\top$ or $\mathtt{F}\alpha$.

Rule (T-RELEASE) models the removal of a VM name α. The premise in the second line verifies that the disposal do not address machines that are executing methods, as discussed in Restriction 3 of Section 2.4.

Rule (T-RETURN) is a bit cryptic. First of all it applies provided $\mathfrak{O} \notin \mathtt{S}$. In fact, by Restriction 4, the type of e in `return` e can be either $\_$ or a virtual machine that has been created by the method. In both cases, these values do not belong to `S`, which is the set of virtual machines in method's arguments – see rule (T-METHOD). In particular, when the type of e is $\_$, by (T-METHOD), $\Gamma(\mathtt{destiny}) = \_$ and, by definition, $\Gamma[\mathfrak{O}' \mapsto \Gamma(\mathfrak{O})] = \Gamma$.

The type system of `vml` is completed with the rules for method declarations and programs, given in Figure 2.6.

Without loss of generality, rule (T-METHOD) assumes that formal parameters of methods are ordered: those of `Int` type occur before those of `Vm` type. We observe that the environment typing the method body binds integer parameters to their same name, while the other ones are bound to fresh VM names (this lets us to have a more precise cost analysis in Section 2.6). We also observe that the returned value $\mathfrak{O}$ may be either $\_$ or a fresh VM name ($\mathfrak{O} \notin \{\alpha\} \cup \overline{\beta}$) as discussed in Restriction 4 of Section 2.4. The constraints in the third line of the premises of (T-METHOD) implement Restriction 1 of Section 2.4. We also observe that $(\Gamma_i(\gamma) = \Gamma_j(\gamma))^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\mathfrak{O})}$ guarantees that every branch of the behavioral type creates a new VM name and, by rule (T-RETURN), the state of the chosen VM name must be always the same.

(T-METHOD)

$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{x}, \overline{\beta}) : \mathfrak{O}, \mathtt{R} \qquad \mathtt{S} = \{\alpha\} \cup \overline{\beta} \qquad \mathfrak{O} \notin \mathtt{S} \\ \Gamma[\mathtt{this} \mapsto \alpha][\mathtt{destiny} \mapsto \mathfrak{O}][\overline{x} \mapsto \overline{x}][\overline{z} \mapsto \overline{\beta}][\alpha \mapsto \varnothing\alpha][\overline{\beta} \mapsto \varnothing\overline{\beta}] \vdash_{\mathtt{S}} s : \mathcal{C}[\mathrm{a}_1 \rhd \Gamma_1] \cdots [\mathrm{a}_n \rhd \Gamma_n] \\ \Big(\Gamma_i(\gamma) = \Gamma_j(\gamma)\Big)^{i,j \in 1..n,\ \gamma \in \mathtt{S} \cup fv(\mathfrak{O})} \qquad \mathtt{R} = (\mathtt{S} \cup fv(\mathfrak{O})) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}\end{array}}{\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{Int}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\}\ : \mathtt{m}\ \alpha(\overline{x}, \overline{\beta})\ \{\mathcal{C}[\mathrm{a}_1 \rhd \varnothing] \cdots [\mathrm{a}_n \rhd \varnothing]\} : \mathfrak{O},\ \mathtt{R}}$$

(T-PROGRAM)

$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}} \qquad \Gamma[\mathtt{this} \mapsto \mathit{start}] \vdash_{\mathrm{start}} s : \mathcal{C}[\mathrm{a}_1 \rhd \Gamma_1] \cdots [\mathrm{a}_n \rhd \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\overline{F\ x\ ;}\ s\} : \overline{\mathbb{C}},\ \mathcal{C}[\mathrm{a}_1 \rhd \varnothing] \cdots [\mathrm{a}_n \rhd \varnothing]}$$

Figure 2.6: Behavioral typing rules of method and programs.

We display behavioral types examples by using codes from Sections 2.2 and 2.5. Actually, the following types do not abstract a lot from codes because the programs of the previous sections have been designed for highlighting the issues of our technique. The following listing shows the behavioral type of `double_release` and the step by step rule application for its calculation.

```
Int double_release(Vm x, Vm y) {
       release x;
       release y;
       return 0 ;
}
```

```
double_release α(β, γ) {  // T-Method
    β✓ ;                  // T-Release
    γ✓ ;                  // T-Release, T-Seq
    0                     // T-Return, T-Seq
} - , {β, γ}
```

The behavioral types of (`fact` and `costly_fact` examples from Section 2.5 are the following (we remove tailing `0` when irrelevant):

```
fact α(n) {
 (n==0){ 0 }
 +(n>0){ νx:fact α(n − 1) ; x✓ }
} - , { }
```

```
costly_fact α(n) {
 (n==0){ 0 }
 +(n>0){ νβ ;
       νx : costly_fact β(n − 1) ;
       x✓ ; β✓ }
} - , { }
```

we notice that the type of `costly_fact` records the order between the recursive invocation and the release of the machine. In the case of the behavioral type of `double_release` the key point is that the releases $\beta^{\checkmark}$ and $\gamma^{\checkmark}$ in `double_release` are conditioned by the values of β and γ when the method is invoked, we keep track of this with the "effects set" $\{\beta,\gamma\}$.

2.6 The analysis of `vml` behavioral types

The types returned by the system in Section 2.5 are used to compute the resource cost of a `vml` program. This computation is performed by a solver of *cost equations*. These cost equations are terms

$$m(\overline{x}) = exp \quad [se]$$

where m is a (cost) function symbol, *exp* is an expression that may contain (cost) function symbols applications, and *se* is a size expression whose variables are contained in $\overline{x}$ (the syntax of *exp* and *se* is given in full detail in [28]).

Basically, our translation maps method types into cost equations, where

- method invocations are translated into function applications,
- virtual machine creations are translated into a `+1` cost,
- virtual machine releases are translated into a `-1` cost,

There are two function calls for every method invocation: one returns the maximal number of resources needed to execute a method `m`, called *peak cost* of `m` and noted $\mathtt{m}_{\text{peak}}$, and the other returns the number of resources the method `m` creates without releasing, called *net cost* of `m` and noted $\mathtt{m}_{\text{net}}$. These functions are used to define the cost of sequential execution and parallel execution of methods. For example, omitting arguments of methods, the peak cost of the sequential composition of two methods `m` and `m'` is the maximal value between $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}} + \mathtt{m}'_{\text{peak}}$; while the cost of the parallel execution of `m` and `m'` is $\mathtt{m}_{\text{peak}} + \mathtt{m}'_{\text{peak}}$.

There are two difficulties that entangle our translation, both related to method invocations: the management of arguments' identities and the management of arguments' values.

2.6.1 Arguments' identities

Consider the following code and the corresponding behavioral types

```
Int simple_release(VM x) {
       release x ;
       return 0 ;
}

Int m(VM x, VM y) {
    Fut<Int> f;
    f = this!simple_release(x);
    release y; f.get;
    return 0;
}
```

$$\begin{array}{l}
\mathtt{simple_release}\ \alpha(\beta)\{ \\
\quad \beta^{\checkmark} \\
\}_{-}, \{\beta\} \\
\\
\mathtt{m}\ \alpha(\beta,\gamma)\{ \\
\quad \nu f : \mathtt{simple_release}\ \alpha(\beta)\, ; \\
\quad \gamma^{\checkmark}\, ; \\
\quad f^{\checkmark} \\
\}_{-}, \{\beta,\gamma\}
\end{array}$$

We notice that, in the type of `m`, there is not enough information to determine whether $\gamma^{\checkmark}$ will have a cost equal to `-1` or `0`. In fact, in the rule (T-METHOD), we assumed that the arguments were pairwise different. However, this is not the case for invocations. For instance, if `m` is invoked with two arguments that are equal – $\beta = \gamma$ – then γ is going to be released by the invocation $\mathtt{free}(\beta)$ and therefore it counts `0`. We solve this problem of arguments' identity in the analysis of behavioral types, in particular in the translation of method types. Namely, the above method `m` is translated in four cost functions: $\mathtt{m}_{\text{peak}}^{\{1\},\{2\}}(x,y)$ and $\mathtt{m}_{\text{net}}^{\{1\},\{2\}}(x,y)$, which correspond to the invocations where $x \neq y$, and $\mathtt{m}_{\text{peak}}^{\{1,2\}}(x)$ and $\mathtt{m}_{\text{net}}^{\{1,2\}}(x)$, which correspond to the

invocations where $x = y$. (The equivalence relation in the superscript never mention `this`, which is also an argument, because, in this case `this` cannot be identified with the other arguments, see below.) Then the translation of an invocation to a method `m` redirects the invocation to $\mathtt{m}^{\Xi}$, where Ξ is the equivalence relation expressing the identity of the arguments.

The function computing the equivalence relation of the arguments of an invocation is `EqRel`. `EqRel` takes a tuple of VM names and returns a partition of the indexes of the tuple:

$$\mathtt{EqRel}(\alpha_0, \cdots, \alpha_n) \;=\; \bigcup_{i \in 0..n} \{\, \{j \mid \alpha_j = \alpha_i\} \,\}$$

That is, if two indexes are in the same set then the corresponding elements in the tuple are equal. So, for instance, $\mathtt{EqRel}(\alpha, \beta, \alpha) = \{\{0, 2\}, \{1\}\}$. We notice that, the possible outputs of $\mathtt{EqRel}(\alpha_0, \alpha_1, \alpha_2)$, without knowing the identities of α_0, α_1, and α_2, are

$\{\{0\}\{1\}, \{2\}\}$	the three arguments are pairwise different
$\{\{0, 1\}, \{2\}\}$	the first and second argument are equal, the third is different
$\{\{0, 2\}, \{1\}\}$	the first and third argument are equal, the second is different
$\{\{0\}, \{1, 2\}\}$	the second and third argument are equal, the first is different
$\{\{0, 1, 2\}\}$	all the arguments are equal

Henceforth, 10 cost functions will correspond to a method with three arguments, 5 for its peak cost and 5 for its net cost.

Next, we also notice that the actual arguments of a $\mathtt{m}^{\{\{0,1\},\{2\}\}}$ are not three but two: in this case the second argument is useless. Therefore we need a notation for selecting the two relevant arguments in these cases. We use the notation (perhaps awkward) $\mathtt{EqRel}(\alpha, \beta, \alpha, \gamma)(\alpha, \beta, \alpha, \gamma)$ that actually returns the triple (α, β, γ).

Without loosing in generality, we will always assume that the canonical representative of a set containing 0 is always 0. This index represents the `this` object and we remind that, by Simplification 3 in Section 2.4, such an object cannot be released. This is the reason why, in the foregoing discussion about the method `m`, we have not mentioned `this`. Additionally, in order to simplify the translation of method invocations, we also assume that the argument `this` is always different from other arguments (the general case just requires more details).

2.6.2 (Re)computing argument's states

In the rules of Figure 2.5, in order to enforce the restrictions in Section 2.4, we have already computed the state of every machine. In this section we

recompute them for a different reason: obtaining a (more) precise cost analysis. Of course one might record the computation of VM states in behavioral types. However, this solution has the drawback that behavioral types become unintelligible because they carry information that is needed at a later stage by the analyzer.

Let a *translation environment*, ranged over Ψ, Ψ', be a mapping from VM names to VM states and from future names to triples $(\Psi', \mathtt{R}, \mathtt{m}\ \beta(\overline{se}, \overline{\beta}) \to \mathfrak{o})$. The translation environment Ψ' in $(\Psi', \mathtt{R}, \mathtt{m}\ \beta(\overline{se}, \overline{\beta}) \to \mathfrak{o})$ is only defined on VM names. For this reason, we call it *vm-translation environment*. We define the following auxiliary functions

– let Ψ be a vm-translation environment. Then

$$\Psi|_X(\alpha) \stackrel{def}{=} \begin{cases} \Psi(\alpha) & \text{if } \alpha \in X \\ \text{undefined} & \text{otherwise} \end{cases}$$

– the *update of a vm-translation environment* Ψ with respect to f and Ψ', written $\Psi \searrow^f \Psi'$, returns a vm-translation environment defined as follows:

$$(\Psi \searrow^f \Psi')(\alpha) \stackrel{def}{=} \begin{cases} (\mathtt{F}'\backslash\{f\})(\mathtt{t} \sqcap \mathtt{t}') & \text{if } \Psi(\alpha) = \mathtt{F}\mathtt{t} \text{ and } \Psi'(\alpha) = \mathtt{F}'\mathtt{t}' \\ \text{undefined} & \text{otherwise} \end{cases}$$

This operation $\Psi \searrow^f \Psi'$ updates the vm-translation environment Ψ that is stored in the future f with the translation environment at the synchronization point. It is worth observing that, by the definition of our type system and the following translation function, the values of $\Psi(\alpha)$ and $\Psi'(\alpha)$ are related. In particular, if $\mathtt{t} = \alpha$ then $\mathtt{t}'$ can be either α or $\alpha\downarrow$ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation on the future f); if $\mathtt{t} = \top$ then $\mathtt{t}'$ can be either $\top$ or ∂ (the machine is released by a method that has been invoked in parallel) or $\bot$ (the machine has been released before the `get` operation).

– the *merge operation*, noted $\Psi(\Delta)$, where Ψ is a vm-translation environment and Δ is an equivalence relation, returns a *substitution* defined as follows. Let

$$\mathtt{t} \otimes^\alpha \mathtt{t}' \stackrel{def}{=} \begin{cases} \bot & \text{if } \mathtt{t} = \bot \text{ or } \mathtt{t}' = \bot \\ \alpha & \text{if } \mathtt{t} \text{ and } \mathtt{t}' \text{ are variables} \\ \alpha\downarrow & \text{otherwise} \end{cases}$$

$$\mathtt{F}_1\mathbb{t}_1 \otimes^\alpha \mathtt{F}_2\mathbb{t}_2 \stackrel{def}{=} (\mathtt{F}_1 \cup \mathtt{F}_2)(\mathbb{t}_1 \otimes^\alpha \mathbb{t}_2)$$

Then

$$\Psi(\Delta) \,:\, \alpha \mapsto \bigotimes^{\Delta(\alpha)} \{\Psi(\beta) \mid \beta \in \mathrm{dom}(\Psi) \text{ and } \Delta(\beta) = \Delta(\alpha)\}$$

for every $\alpha \in \mathrm{dom}(\Psi)$.

The operator $\otimes^\alpha$ has not been defined on VM values as ∂ or $\top$ because we merge VM names whose image by Ψ are either $\mathtt{F}\beta$ or $\mathtt{F}\beta\!\downarrow$ or $\mathtt{F}\bot$. As a notational remark, we observe that $\Psi(\Delta)$ is noted as a map $[\alpha_1 \mapsto \mathtt{F}_1\mathbb{t}_1, \cdots, \alpha_n \mapsto \mathtt{F}_n\mathbb{t}_n]$ instead of the standard notation $\left[\mathtt{F}_1\mathbb{t}_1, \cdots, \mathtt{F}_n\mathbb{t}_n/_{\alpha_1, \ldots, \alpha_n}\right]$. These two notations are clearly equivalent: we prefer the former one because it will let us to write $\Psi(\Delta)(\alpha)$ or even $\Psi(\Delta)(\alpha_1, \cdots, \alpha_n)$ with the obvious meanings.

To clarify the reason for a merge operator, consider the atom $f^\checkmark$ within a behavioral type that binds f to $\mathtt{foo}\,\alpha(\beta, \gamma)$. Assume to evaluate this type with $\Delta = \{\{\beta, \gamma\}\}$. That is, the two arguments are actually identical. Which are the values of β and γ for evaluating $\mathtt{foo}^\Delta_{\mathrm{peak}}$ and $\mathtt{foo}^\Delta_{\mathrm{net}}$? Well, we have

1. to select the representative between β and γ: it will be $\Delta(\beta)$ – which is equal to $\Delta(\gamma)$;

2. to take a value that is smaller than $\Psi(\beta)$ and $\Psi(\gamma)$ (but greater than any other value that is smaller);

3. to substitute β and γ with the result of 2.

For instance, let $\Psi = [\alpha \mapsto \varnothing\alpha, \beta \mapsto \varnothing\beta\!\downarrow, \gamma \mapsto \varnothing\gamma]$ and $\Delta(\beta) = \Delta(\gamma) = \beta$. We expect that a value for the item 2 above is $\varnothing\beta\!\downarrow$ and the substitution of the item 3 is $[\varnothing\beta\!\downarrow, \varnothing\beta\!\downarrow/_{\beta, \gamma}]$. Formally, the operation returning the value for 2 is $\otimes^\beta$ and the substitution of item 3 is the output of the merge operation.

2.6.3 The translation function

The translation function, called `translate`, is structured in three parts that respectively correspond to simple atoms, behavioral types, and method types and full programs. This function carries five arguments:

1. Δ is the *equivalence relation on formal parameters* identifying those that are equal. We assume that $\Delta(x)$ returns the unique representative of the equivalence class of x. For simplicity we also let $\Delta(x) = x$ for every x that belongs to the local variables. Therefore we can use Δ also as a substitution operation.

2. Ψ is the *translation environment* which stores temporary information about futures that are active (unsynchronized) and about the state of VM names;

3. α is the name of the virtual machine of the current behavioral type;

4. $\overline{e}$ is the sequence of (over-approximated) costs of the current execution branch;

5. the behavioral type being translated; it may be either $\mathbb{a}$, $\mathbb{c}$ or $\overline{\mathbb{C}}$.

In the definition of `translate` we use the two functions

$$\texttt{CNEW}(\alpha) = \begin{cases} \texttt{0} & \alpha = \bot \\ \texttt{1} & \text{otherwise} \end{cases} \qquad \texttt{CREL}(\alpha) = \begin{cases} \texttt{-1} & \alpha = \top \\ \texttt{0} & \text{otherwise} \end{cases}$$

The left-hand side function is used when a virtual machine is created. It returns `1` or `0` according to the virtual machine that is executing the code can be alive ($\alpha \neq \bot$) or not, respectively. The right-hand side function is used when a virtual machine is released (in correspondence of atoms $\beta^{\checkmark}$). The release is effectively computed – value `-1` – only when the virtual machine that is executing the code is alive ($\alpha = \top$).

We will assume the presence of a lookup function `lookup` that takes method invocations $\texttt{m}\ \alpha(\overline{\mathbb{r}}, \overline{\beta})$ and returns tuples $\mathbb{c} : \mathbb{o}, \texttt{R}$. This function is left unspecified. We also write $\texttt{R}[{}^{\mathbb{o}'}/_{\mathbb{o}}]$ to denote a set that is equal to `R` if $\mathbb{o} \notin \texttt{R}$, that is equal to $(\texttt{R} \backslash \{\mathbb{o}\}) \cup \{\mathbb{o}'\}$, otherwise.

The definition of `translate` follows. We begin with the translation of atoms.

$\texttt{translate}[\Delta, \Psi, \alpha](\overline{e}; e)(\mathbb{a}) =$

$$\begin{cases} (\Psi, \overline{e}; e) & \text{when } \mathbb{a} = \texttt{0} \\ (\Psi[\beta \mapsto \varnothing\top], \overline{e}; e; e + \texttt{CNEW}(\mathbb{t})) & \text{when } \mathbb{a} = \nu\beta \quad \text{and} \quad \Psi(\alpha) = \texttt{F}\mathbb{t} \\ (\Psi[\Delta(\beta) \mapsto \varnothing\bot], \overline{e}; e; e + \texttt{CREL}(\mathbb{t})) & \text{when } \mathbb{a} = \beta^{\checkmark} \quad \text{and} \quad \Psi(\Delta(\beta)) = \texttt{F}\mathbb{t} \\ (\Psi'[f \mapsto (\Psi|_{\Delta(\beta,\overline{\beta})}, \texttt{R}, \texttt{m}\ \Delta(\beta)(\overline{\mathbb{r}}, \Delta(\overline{\beta}) \to \mathbb{o}))], \overline{e}; e; e + f) & \text{when } \mathbb{a} = \nu f : \texttt{m}\ \beta(\overline{\mathbb{r}}, \overline{\beta}) \\ & \text{and } \Psi' = \Psi[\beta \mapsto (\texttt{F} \cup \{f\})\mathbb{t}]^{\beta \in \texttt{R}, \Psi(\beta) = \texttt{F}\mathbb{t}} \\ & \text{and } \texttt{lookup}(\texttt{m}\ \Delta(\beta)(\overline{se}, \Delta(\overline{\beta}))) = \mathbb{c} : \mathbb{o}, \texttt{R} \\ (\Psi'' \backslash f,\ (\overline{e}; e)\sigma; (e)\sigma' + \sum_{\gamma \in \Delta(\texttt{R}), \Theta(\gamma) = \texttt{F}\mathbb{t}, \texttt{F} \neq \varnothing} \texttt{CREL}(\mathbb{t})) & \text{when } \mathbb{a} = f_{\mathbb{o}'}{}^{\checkmark} \text{ and } \Psi(f) = (\Psi', \texttt{R}, \texttt{m}\ \beta(\overline{\mathbb{r}}, \overline{\beta}) \to \mathbb{o}) \\ & \text{and } \Theta = \Psi' \swarrow^{f}_{\searrow} \Psi \quad \text{and} \quad \Theta(\texttt{EqRel}(\beta, \overline{\beta})) = (\texttt{F}_1\mathbb{t}_1, \cdots, \texttt{F}_n\mathbb{t}_n) \\ & \text{and } \sigma = [\texttt{m}^{\Xi}_{\text{peak}}(\overline{\mathbb{r}}, \texttt{F}_1\mathbb{t}_1\ \Downarrow, \cdots, \texttt{F}_n\mathbb{t}_n\ \Downarrow)\ \Downarrow)/f] \\ & \text{and } \sigma' = [\texttt{m}^{\Xi}_{\text{net}}(\overline{\mathbb{r}}, \texttt{F}_1\mathbb{t}_1\ \Downarrow, \cdots, \texttt{F}_n\mathbb{t}_n\ \Downarrow)/f] \\ & \text{and } \texttt{R}' = \texttt{R}[{}^{\mathbb{o}'}/_{\mathbb{o}}] \\ & \text{and } \Psi'' = \Psi[\gamma \mapsto \varnothing\bot]^{\gamma \in \texttt{R}'}[\gamma' \mapsto \Theta(\beta)]^{\gamma' \in fv(\mathbb{o}') \backslash \texttt{R}'} \end{cases}$$

In the definition of `translate` we always highlight the last expression in the sequence of costs of the current execution branch (the fourth input).

This is because the cost of the parsed atom applies to it, except for the case of $f^{\checkmark}$. In this last case, let $\overline{e}; e$ be the expression. Since the atom expresses the synchronization of f, $\overline{e}; e$ will have occurrences of f. In this case, the function `translate` has to compute two values: the maximum number of resources used by (the method corresponding to) f during its execution – the *peak cost* used in the substitution σ – and the resources used upon the termination of (the method corresponding to) f – the *net cost* used in the substitution σ'. In particular, this last value has to be decreased by the number of resources released by the method. This is the purpose of the addend $\sum_{\gamma \in \Delta(\mathtt{R}), \Theta(\gamma) = \mathtt{F}\Bbbt, \mathtt{F} \neq \varnothing} \mathtt{CREL}(\Bbbt)$ that removes machines that are going to be removed by parallel methods (the constraint $\mathtt{F} \neq \varnothing$) because the other ones have been already counted both in the peak cost and in the net cost. We observe that the instances of the method $\mathtt{m}_{\text{peak}}$ and $\mathtt{m}_{\text{net}}$ that are invoked are those corresponding to the equivalence relation of the tuple $(\beta, \overline{\beta})$.

The translation of behavioral types is given by composing the definitions of the atoms. In this case, the output of `translate` is a set of cost equations.

$\mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{c}) \;=$

$$\begin{cases} \{(se')\{\,\overline{e'}\,\}\} & \text{when } \mathbb{c} = \mathbb{a} \rhd \varnothing \text{ and } \mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{a}) = (\Psi', (se')\{\,\overline{e'}\,\}) \\ C'' & \text{when } \mathbb{c} = \mathbb{a} \fatsemi \mathbb{c}' \text{ and } \mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{a}) = (\Psi', \{(se')\{\,\overline{e'}\,\}\}) \\ & \text{and } \mathrm{dom}(\Psi') \backslash \mathrm{dom}(\Psi) = \mathtt{S} \text{ and } \mathtt{translate}(\Delta \cup \{\mathtt{S}\}, \Psi', \alpha, (se')\{\,\overline{e'}\,\}, \mathbb{c}') = (\Psi'', C'') \\ C' \cup C'' & \text{when } \mathbb{c} = \mathbb{c}_1 + \mathbb{c}_2 \text{ and } \mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{c}_1) = (\Psi', C') \\ & \text{and } \mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{c}_2) = (\Psi'', C'') \\ C' & \text{when } \mathbb{c} = (se')\{\mathbb{c}'\} \text{ and } \mathtt{translate}(\Delta, \Psi, \alpha, (se \wedge se')\{\,\overline{e}\,\}, \mathbb{c}') = (\Psi', C') \\ C' & \text{when } \mathbb{c} = (e')\{\mathbb{c}'\} \text{ and } e' \text{ contains } \_ \text{ and } \mathtt{translate}(\Delta, \Psi, \alpha, (se)\{\,\overline{e}\,\}, \mathbb{c}') = (\Psi', C') \end{cases}$$

The translation of method types and behavioral type programs is given below. Let $\mathcal{P}$ be the set of partitions of $1..n$. Then

$$\mathtt{translate}(\mathtt{m}\ \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\,\mathbb{c}\,\} : \mathbb{D},\ \mathtt{R}) = \bigcup_{\Xi \in \mathcal{P}} \mathtt{translate}(\Xi, \mathtt{m}\ \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\,\mathbb{c}\,\} : \mathbb{D},\ \mathtt{R})$$

where $\mathtt{translate}(\Xi, \mathtt{m}\ \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_n)\ \{\,\mathbb{c}\,\} : \mathbb{D},\ \mathtt{R})$ is defined as follows. Let

$$\Delta = \{\,\{\,\alpha_{i_1}, \ldots, \alpha_{i_m}\,\} \mid \{\,i_1, \ldots, i_m\,\} \in \Xi\,\} \quad \text{and} \quad [\Delta] = \{\alpha \mapsto \varnothing\alpha \mid \alpha = \Delta(\alpha)\}$$

$$\text{and} \quad \mathtt{translate}(\Delta, [\Delta], \alpha_1)(0)(\mathbb{c}) = \bigcup_{i=1}^{n} (se_i)\{\,e_{1,i}; \ldots; e_{h_i,i}\,\}\ .$$

Then

$$\texttt{translate}(\Xi, \texttt{m}\ \alpha_1(\overline{x}, \alpha_2, \ldots, \alpha_k)\ \{\,\mathbb{c}\,\} : \mathbb{D},\ \texttt{R}) \ =$$

$$\begin{cases}
\texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = 0 & [\alpha_1 = \bot] \\
\texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{1,1} & [se_1 \wedge \alpha_1 \neq \bot] \\
\quad\vdots & \\
\texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_1,1} & [se_1 \wedge \alpha_1 \neq \bot] \\
\texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{1,2} & [se_2 \wedge \alpha_1 \neq \bot] \\
\quad\vdots & \\
\texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_n,n} & [se_n \wedge \alpha_1 \neq \bot] \\
\texttt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = 0 & [\alpha_1 = \bot] \\
\texttt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = \texttt{m}^{\Xi}_{\text{peak}}(\overline{x}, \Xi[\alpha_1, \ldots, \alpha_n]) & [\alpha_1 = \partial] \\
\texttt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_1,1} & [se_1 \wedge \alpha_1 = \top] \\
\quad\vdots & \\
\texttt{m}^{\Xi}_{\text{net}}(\overline{x}, \Xi[\alpha_1, \alpha_2, \ldots, \alpha_k]) = e_{h_n,n} & [se_n \wedge \alpha_1 = \top]
\end{cases}$$

Let $(\mathbb{C}_1\ \ldots\ \mathbb{C}_n,\ \mathbb{c})$ be a behavioral type program and let

$$\texttt{translate}(\varnothing, \varnothing, \alpha, (\texttt{true})\{\,0\,\}, \mathbb{c}) = \bigcup_{j=1}^{m} (se_j)\{\,e_{1,j}; \cdots ; e_{h_j,j}\,\}$$

then

$$\texttt{translate}(\mathbb{C}_1\ \ldots\ \mathbb{C}_n,\ \mathbb{c}) \ = \ \begin{cases}
\texttt{translate}(\mathbb{C}_1) \cdots \texttt{translate}(\mathbb{C}_n) & \\
\texttt{main}() = 1 + e_{1,1} & [se_1] \\
\quad\vdots & \\
\texttt{main}() = 1 + e_{h_1,1} & [se_1] \\
\texttt{main}() = 1 + e_{1,2} & [se_2] \\
\quad\vdots & \\
\texttt{main}() = 1 + e_{h_m,m} & [se_m]
\end{cases}$$

As an example, we show the output of `translate` when applied to the behavioral type of `double_release` computed in Section 2.5. Since `double_release` has two arguments, we generate two sets of equations, as discussed above. In order to ease the reading, we omit the equivalence classes of arguments that label function names: the reader can grasp them from the number of arguments. For the same reason, we represent a partition $\{\{1\}, \{2\}, \{3\}\}$ corresponding to VM names α_1, α_2 and α_3 by $[\alpha_1, \alpha_2, \alpha_3]$ and $\{\{1\}, \{2, 3\}\}$ by $[\alpha_1, \alpha_2]$ (we write the canonical representatives). For simplicity we do not add the partition to the name of the method.

$$\texttt{translate}([\alpha_1,\alpha_2,\alpha_3], \mathit{double\_release}\ \alpha_1(\alpha_2,\alpha_3)\ \{\mathbb{c}\} : \_,\ \{\alpha_2,\alpha_3\}) =$$

$$\begin{cases}
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 \neq \bot] \\
\\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = \mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2,\alpha_3) & [\alpha_1 = \partial] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2,\alpha_3) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\alpha_3) & [\alpha_1 = \top]
\end{cases}$$

$$\texttt{translate}([\alpha_1,\alpha_2], \mathit{double\_release}\ \alpha_1(\alpha_2)\ \{\mathbb{c}\} : \_,\ \{\alpha_2\}) =$$

$$\begin{cases}
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) & [\alpha_1 \neq \bot] \\
\mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 \neq \bot] \\
\\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = 0 & [\alpha_1 = \bot] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = \mathit{double_release}_{\text{peak}}(\alpha_1,\alpha_2) & [\alpha_1 = \partial] \\
\mathit{double_release}_{\text{net}}(\alpha_1,\alpha_2) = \texttt{CREL}(\alpha_2) + \texttt{CREL}(\bot) & [\alpha_1 = \top]
\end{cases}$$

To highlight a cost computation concerning `double_release`, consider the following two potential users and the corresponding behavioral types

```
Int user1() {
     Vm x ; Vm y ; Fut<Int> f ;
     x = new Vm ; y = new Vm;
     f = this!double_release(x, y);
     f.get ; return 0 ;
}
```

```
Int user2() {
     Vm x ; Fut<Int> f ;
     Vm x = new Vm ;
     f = this!double_release(x, x);
     f.get ; return 0 ;
}
```

$$\texttt{user1}\ \alpha(\,)\{\ \ \nu\beta \fatsemi \nu\gamma \fatsemi \nu\, f : \texttt{double\_release}\,\alpha(\beta,\gamma) \fatsemi f^{\checkmark}\ \}\ \texttt{-}\ ,\ \{\ \}$$

$$\texttt{user2}\ \alpha(\,)\{\ \ \nu\beta \fatsemi \nu\, f : \texttt{double\_release}\,\alpha(\beta,\beta) \fatsemi f^{\checkmark}\ \}\ \texttt{-}\ ,\ \{\ \}$$

The translations of the foregoing types give the following set of equations

$$\texttt{translate}([\alpha_1], user1\ \alpha_1()\ \{\mathbb{c}_{user1}\} : \_,\ \{\}) =$$

$$\begin{cases}
user1_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
user1_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
user1_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
user1_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
user1_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \ double_release_{\text{peak}}(\alpha_1, \top, \top) & [\alpha_1 \neq \bot] \\
user1_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \ double_release_{\text{net}}(\alpha_1, \top, \top) & [\alpha_1 \neq \bot] \\
\\
user1_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
user1_{\text{net}}(\alpha_1) = user1_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
user1_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + \texttt{CNEW}(\alpha_1) + \ double_release_{\text{net}}(\alpha_1, \top, \top) & [\alpha_1 = \top]
\end{cases}$$

$$\texttt{translate}([\alpha_1], user2\ \alpha_1()\ \{\mathbb{c}_{user2}\} : \_,\ \{\}) =$$

$$\begin{cases}
user2_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
user2_{\text{peak}}(\alpha_1) = 0 & [\alpha_1 \neq \bot] \\
user2_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) & [\alpha_1 \neq \bot] \\
user2_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + double_release_{\text{peak}}(\alpha_1, \top) & [\alpha_1 \neq \bot] \\
user2_{\text{peak}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + double_release_{\text{net}}(\alpha_1, \top) & [\alpha_1 \neq \bot] \\
\\
user2_{\text{net}}(\alpha_1) = 0 & [\alpha_1 = \bot] \\
user2_{\text{net}}(\alpha_1) = user2_{\text{peak}}(\alpha_1) & [\alpha_1 = \partial] \\
user2_{\text{net}}(\alpha_1) = \texttt{CNEW}(\alpha_1) + double_release_{\text{net}}(\alpha_1, \top) & [\alpha_1 = \top]
\end{cases}$$

If we compute the cost of $user1_{\text{peak}}(\alpha)$ and $user1_{\text{net}}(\alpha)$ we obtain `2` and `0`, respectively. That is, in this case, `double_release` being invoked with two different arguments has cost `-2`. On the contrary, the cost of $user2_{\text{peak}}(\alpha)$ and $user2_{\text{net}}(\alpha)$ is `1` and `0`, respectively. That is, in this case, `double_release` being invoked with two equal arguments has cost `-1`.

2.7 Correctness

The correctness of our technique is expressed by the following theorem that relates the `vml` language, its behavioral type system and the *cost* of a `vml` program.

Theorem 7.1 (Correctness)**.** *Let* $\overline{M}\ \{\overline{F\ z\ ;}\ s'\}$ *be a well-typed program and let* $\overline{\mathbb{C}}$, $\mathbb{c}$ *be its behavioral type and cn be its initial configuration. Let also n be a solution of the function* $\texttt{translate}(\overline{\mathbb{C}}, \mathbb{c})$*. Then, for every* $cn \longrightarrow^* cn'$, $\texttt{alive}(cn') \leqslant n$.

The proof of this theorem, and therefore of the correctness of our approach, consists of two parts. The first part addresses the correctness of the type system in Section 2.5, which is usually expressed by a subject reduction

theorem. This theorem states that if a configuration cn of the operational semantics is well typed and $cn \rightarrow cn'$ then cn' is well-typed as well. It is worth observing that we cannot hope to demonstrate a statement guaranteeing type-preservation because our types are "behavioral" and change during the evolution of the systems. However, it is critical for the correctness of the cost analysis that there exists a relation between the type of cn and the type of cn'.

Therefore, a subject reduction for the type system of Section 2.5 requires

1. the extension of the typing to configurations;
2. the definition of an evaluation relation $\rightsquigarrow$ between behavioral types.

2.7.1 The extension of the typing to configurations

When typing configurations there are two kinds of vm names: *static-time VM names* and *runtime VM names*. The former names are ranged over by α, β, $\cdots$; the latter names, which also include err, are ranged over by o, o', $\cdots$. With an abuse of notation, in the following, α and $\mathbb{o}$ will also range over runtime VM names o and err. This abuse will let us to reuse the typing rules in Figure 2.5 (with one exception, see below). The syntax of runtime types is:

$$
\begin{array}{lcll}
\mathbb{k} & ::= & \mathtt{0} \mid f(\bot) \mid f(\_) \mid f(o) \mid (f, \mathtt{m}\ o(\overline{\mathbb{r}})) \mid vm(o, a \mid \mathbb{f}; \overline{\mathbb{f}}) \mid \mathbb{k} \parallel \mathbb{k} & \text{runtime types} \\
\mathbb{f} & ::= & (f, \mathbb{c}, \mathtt{S}) & \text{process types} \\
\mathbb{a} & ::= & \ldots \mid [\mathbb{o}] & \text{extended atom}
\end{array}
$$

The syntactic category $\mathbb{k}$ models behavioral types only existing at runtime, in particular, virtual machines, futures, and processes. The term $\mathtt{0}$ correspond to an element of the running program that does not have any effect on its cost (*e.g.*, ϵ); $f(\bot)$, $f(\_)$, and $f(o)$ type a future binder: f is the name of the future, $\bot$ or $\_$ or o identify the value of the future, namely either $\bot$, if it is still not computed, or $\_$ or o, if it has been already computed (see the rest of the paragraph for more details); $(f, \mathtt{m}\ o(\overline{\mathbb{r}}))$ corresponds to an invocation message. The type $vm(o, a \mid \mathbb{f}; \overline{\mathbb{f}})$ corresponds to a virtual machine (alive if $a = \top$, dead otherwise) executing the process $\mathbb{f}$ and with a set of processes $\overline{\mathbb{f}}$ to execute. Processes are triples $(f, \mathbb{c}, \mathtt{S})$ where:

- f is the identity of the process (i.e. the name of its future);
- $\mathbb{c}$ is the type of the statement the process will execute;

– $\mathtt{S}$ is a set of runtime VM names.

The term $\Bbbk \parallel \Bbbk$ types the parallel composition of runtime configurations. Atoms are extended with the term $[\circledcirc]$ that represents the returned value of a method – see the discussion below.

The runtime type system consists of rules in Figure 2.5 plus the rules in Figure 2.7. The reader may notice that the first rule (BT-Return) replaces the definition of the judgment $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c} \rhd \Gamma'$ for the `return` statement in Figure 2.5. This modification, together with the extensions of atoms, is used to refine the (static-time) typing rules in order to deal with a naming issue in the subject reduction theorem. In particular, when a method returns a freshly created virtual machine, we associate to it a fresh name in (T-Invoke) that we can use later in (T-Get) and in the rest of the typing of the (static) program. The problem is that, when we type the method invocation (i.e. when we collect the information about the method and its return value in order to type future usage of this value), the identity of the returned virtual machine is still unknown because the virtual machine has not yet been created. In order to solve this problem, we extend atoms with the term $[\circledcirc]$, which retains the returned value and we use the new rule (BT-Return) to type the `return` statements.

The other rule of Figure 2.7 that we comment is (BT-Vm). This rule allows one to type a VM term containing a number of processes p. To this aim, one has to type every p by means of the judgment $\Gamma[\mathtt{this} \mapsto o] \vdash_{\mathtt{S}} p : (f, \mathbb{c}, \mathtt{S})$; namely the index $\mathtt{S}$ and the third argument of the process type must be the same. In fact, this argument records the actual VM names of the corresponding method invocation – see rule (R-Bind-Mtd) in Figure 2.8. In turn, this set is used in (BT-Process) to type the statements of the processes.

Lemma 7.2 (Substitution Lemma)**.** *Let* $\Gamma \vdash cn : \Bbbk$ *and let* ι *be an injective renaming of names, VM names, and future names. Let* $\iota(\Gamma) \stackrel{def}{=} [\iota(x) \mapsto \iota(\Gamma(x))]^{x \in \mathrm{dom}(\Gamma)}$.

Then $\iota(\Gamma) \vdash \iota(cn) : \iota(\Bbbk)$. *Similarly for* $\Gamma \vdash_{\mathtt{S}} \{l \,|\, s\} : (f, \mathbb{c})$ *and* $\Gamma \vdash_{\mathtt{S}} s : \mathbb{c} \rhd \Gamma'$ *(in these two cases also* $\mathtt{S}$ *is mapped by* ι*) and for* $\Gamma \vdash e : \mathbb{x}$.

Proof. The proof proceeds straightforward by induction on the structure of s. □

2.7.2 Runtime types evolution

As usual with behavioral types, our types change while the programs execute. These changes can be defined and, more importantly, can be related to the

$$\frac{\Gamma \vdash e : \mathfrak{O} \qquad \Gamma \vdash \texttt{destiny} : \mathfrak{O}' \qquad \mathfrak{O} \notin \mathtt{S}}{\Gamma \vdash_{\mathtt{S}} \texttt{return}\ e : [\mathfrak{O}] \rhd \Gamma[\mathfrak{O}' \mapsto \Gamma(\mathfrak{O})]}\ \text{(BT-Return)}$$

$$\frac{\Gamma(f) = (\mathfrak{O}, o, \mathtt{F}\mathbb{t}, \mathtt{R})}{\Gamma \vdash \mathit{fut}(f, \bot) : f(\bot)}\ \text{BT-Fut-Running}$$

$$\frac{\Gamma \vdash v : \mathfrak{O} \qquad \Gamma \vdash f : \mathfrak{O}}{\Gamma \vdash \mathit{fut}(f, v) : f(\mathfrak{O})}\ \text{BT-Fut-Computed}$$

$$\frac{\Gamma(f) = (\mathfrak{O}, o, \mathtt{F}\mathbb{t}, \mathtt{R}) \qquad v \neq \bot \qquad \Gamma \vdash v : \mathfrak{O}' \qquad \alpha \in \mathtt{R}[{}^{\mathfrak{O}'}/_{\mathfrak{O}}] \Rightarrow \Gamma(\alpha) = \varnothing\bot}{\Gamma \vdash \mathit{fut}(f, v) : f(\mathfrak{O}')}\ \text{BT-Fut-Unsync}$$

$$\Gamma \vdash \epsilon : \mathtt{0}\ \text{BT-Empty}$$

$$\frac{\Gamma \vdash \mathtt{m}\, o(\overline{v}) : \mathfrak{O}, \mathtt{R} \qquad \Gamma(f) = (\mathfrak{O}, o, \mathtt{F}\mathbb{t}, \mathtt{R}) \qquad \Gamma \vdash \overline{v} : \overline{\mathbb{r}}}{\Gamma \vdash \mathit{invoc}(o, f, \mathtt{m}, \overline{v}) : (f, o\ \mathtt{m}(\overline{\mathbb{r}}))}\ \text{BT-Invoke}$$

$$\frac{\Gamma \vdash o : \mathtt{F}a \qquad (\Gamma \vdash_{\mathtt{S}_i} p_i : \mathbb{f}_i \qquad \mathbb{f}_i = (f_i, \mathbb{c}_i, \mathtt{S}_i))^{i \in 1..n}}{\Gamma \vdash \mathit{vm}(o, a, p_1; \{p_2\ \ldots\ p_n\}) : \mathit{vm}(o, a \mid \mathbb{f}_1; \{\mathbb{f}_2, \ldots, \mathbb{f}_n\})}\ \text{BT-Vm}$$

$$\frac{\Gamma \vdash \mathit{cn} : \mathbb{k} \qquad \Gamma \vdash \mathit{cn}' : \mathbb{k}'}{\Gamma \vdash \mathit{cn}\ \mathit{cn}' : \mathbb{k} \parallel \mathbb{k}'}\ \text{BT-Parallel}$$

$$\frac{\begin{array}{c} l = [(x_i \mapsto v_i)^{i \in I}, \text{destiny} \mapsto f] \qquad (\Gamma \vdash v_i : \mathbb{x}_i)^{i \in I} \qquad \Gamma(f) = (\mathfrak{O}, o, \mathtt{F}\mathbb{t}, \mathtt{R}) \\ \Gamma[(x_i \mapsto \mathbb{x}_i)^{i \in I}][\texttt{destiny} \mapsto \mathfrak{O}] \vdash_{\mathtt{S}} s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1] \cdots [\mathbb{a}_n \rhd \Gamma_n] \\ \left(\Gamma_i(\gamma) = \Gamma_j(\gamma)\right)^{i,j \in 1..n,\ \gamma \in \mathtt{S}} \qquad \mathtt{R} = (\mathtt{S} \cup \mathit{fv}(\mathfrak{O})) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\} \end{array}}{\Gamma \vdash_{\mathtt{S}} \{l \mid s\} : (f, \mathcal{C}[\mathbb{a}_1 \rhd \varnothing] \cdots [\mathbb{a}_n \rhd \varnothing], \mathtt{S})}\ \text{BT-Process}$$

Figure 2.7: Runtime Typing rules

cost analysis. The modifications of types, noted $\rightsquigarrow$, is defined as a reduction relation in Figure 2.8. The rules are mostly an adaptation of the operational semantics of `vml` at the type level. Therefore, to ease the understanding of the relation $\rightsquigarrow$, we use similar names for similar rules. Nevertheless, there are few differences from the operational semantics of `vml`: *i*) rule (ASSIGN) has no corresponding rule in the definition of $\rightsquigarrow$ (assignments are managed at the type system level); *ii*) rule (ACTIVATE) is replaced by the rule (R-GC) (there are no scheduling policies in our runtime types, so the only effect of the rule (ACTIVATE) at the type level is the deletion of a finished process); iii) similarly, rule (ACTIVATE-GET) has no corresponding rule in the definition of $\rightsquigarrow$; and iv) because at runtime all objects identities are known, the rules (READ-FUT), (ASYNC-CALL), (BIND-MTD) and (NEW-VM) have been adjusted in order to replace the names generated a static time with the objects' actual identities.

2.7.3 The subject reduction theorem

Based on the previous construction, we present the subject reduction theorem. Lemma 7.3 relates the static time type system to the runtime type system: this corresponds to the first step of the subject reduction property.

Lemma 7.3. *Let* $P = \overline{M}\ \{\overline{F\ x\ ;}\ s\}$ *be a* `vml` *program, and let* $\Gamma \vdash P : \overline{\mathbb{C}}, \mathbb{c}$. *Then*

$$\Gamma[start \mapsto \top][f_{start} \mapsto (\_, start, \varnothing\top, \varnothing)] \vdash$$
$$vm(start, \top, \{[destiny \mapsto f_{start}] \mid s\}, \varnothing) : vm(start, \top \mid (f_{start}, \mathbb{c}, \varnothing); \varnothing)\ .$$

Proof. The proof follows by the application of rules (BT-VM) and (BT-PROCESS), and using the hypothesis. □

The main subject reduction theorem states that any well-typed runtime program reduces into another well-typed runtime program.

Theorem 7.4 (Subject Reduction)**.** *Let* $\Gamma \vdash cn : \Bbbk$. *If* $cn \rightarrow cn'$ *then there exists* Γ' *such that* $\Gamma' \vdash cn' : \Bbbk'$ *and* $\Bbbk \rightsquigarrow^* \Bbbk'$.

Proof. As usual for Subject Reduction theorems we proceed by case on the reduction rule used in $cn \rightarrow cn'$. Since our types are behavioral we cannot assume that the types of a runtime programs state and its reduction are equivalent, instead we prove that this types $(\Bbbk, \Bbbk')$ are related according to the operational semantics for runtime types, see Figure 2.8. We present the proof of some cases, the rest are either straightforward or similar to those below. Whenever the construction of ι is omitted it is considered as

the identity function. Finally, if not stated otherwise, we implicitly use the notation used in the considered reduction rule:

- Case (Assign).

$$\frac{v = [\![e]\!]_l}{vm(o, \top, \{l \mid x = e; s\}, q) \to vm(o, \top, \{l[x \mapsto v] \mid s\}, q)}\ (\textsc{Assign})$$

  By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e; s\}, q) : \Bbbk$. By application of the corresponding typing rules we have that $\Bbbk = vm(o, \top \mid (f, \mathtt{0}\fatsemi\Cbbb, \mathtt{S}),\ \overline{\mathbb{f}})$. Let $\Gamma' = \Gamma$, then by application of rules (BT-VM) and (BT-Process), and by hypothesis we have that $\Gamma' \vdash vm(o, \top, \{l[x \mapsto v] \mid s\}, q) : \Bbbk'$, with $\Bbbk' = vm(o, \top \mid (f, \Cbbb, \mathtt{S}),\ \overline{\mathbb{f}})$. We notice that $\Bbbk$ and $\Bbbk'$ are related by rule (R-Skip)

- Case (Read-Fut).

$$\frac{f = [\![e]\!]_l \quad v \neq \bot}{vm(o, \top, \{l \mid x = e.\mathtt{get}; s\}, q) \quad fut(f, v) \to vm(o, \top, \{l \mid x = v; s\}, q) \quad fut(f, v)}\ (\textsc{Read-Fut})$$

  By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e.\mathtt{get}; s\}, q) \quad fut(f, v) : \Bbbk$. Let consider the case in which $\Gamma(f) = (\mathfrak{o}, \alpha, \mathtt{F}\mathbb{t}, \mathtt{R})$ (the other case, where $\Gamma(f) = \mathfrak{o}$, is trivial). By reconstruction of the proof tree of the hypothesis, with $l = \{x_i \mapsto \mathbb{x}_i \mid i \in I\}$, we derive that $\Bbbk = vm(o, \top \mid (f', f^{\checkmark}_{\mathfrak{o}}\fatsemi\Cbbb, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\mathfrak{O}'')$ and that $\Gamma[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathfrak{O}'][\beta \mapsto \varnothing\bot]^{\beta\in\mathtt{R}}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta'\in\mathtt{R}'}[f \mapsto \mathfrak{o}] \vdash s : \Cbbb$ with $\mathtt{R}' = fv(\mathfrak{o})\backslash\mathtt{R}$ and $\mathbb{t}' = \mathbb{t} \sqcap (\Gamma(\alpha)\Downarrow)$.

  Let $\iota = [{}^{\mathfrak{O}''}/_{\mathfrak{O}}]$: we have that $\iota(\Gamma) = \Gamma$. Let $\Gamma' = \Gamma[f \mapsto \mathfrak{O}'']$, by lemma 7.2 we have that $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathfrak{O}'][\beta \mapsto \varnothing\bot]^{\beta\in\iota(\mathtt{R})}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta'\in\mathtt{R}''} \vdash s : \iota(\Cbbb)$ with $\mathtt{R}'' = fv(\mathfrak{O}'')\backslash\iota(\mathtt{R})$. To prove the theorem in this case, we need to show that s is can be typed with the environment $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathfrak{O}']$. First, because of the side condition in rule (BT-Fut-Unsync), we have that $\Gamma'(\beta) = \varnothing\bot$ for all $\beta \in \iota(\mathtt{R})$. We thus have

$$\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathfrak{O}'][\beta \mapsto \varnothing\bot]^{\beta\in\iota(\mathtt{R})}[\beta' \mapsto \varnothing\mathbb{t}']^{\beta'\in\mathtt{R}''} =$$

$$\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathfrak{O}'][\beta' \mapsto \varnothing\mathbb{t}']^{\beta'\in\mathtt{R}''}$$

  Now, to remove the last substitution $[\beta' \mapsto \varnothing\mathbb{t}']^{\beta'\in\mathtt{R}''}$, we have four cases:

  – either $\mathtt{R}''$ is empty, in which case that substitution is the identity;

- either $\mathbb{t}'$ is $\top$, in which case, by construction of the set $\mathtt{R}$, we have $\Gamma'(\beta') = \Gamma(\beta) = \top$: the substitution does not change Γ';
- either $\mathbb{t}'$ is ∂ and $\Gamma'(\beta')$ is $\top$, in which case $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathbb{o}']$ $\vdash s : \iota(\mathbb{c})$ holds;
- either $\mathbb{t}'$ is ∂ and $\Gamma'(\beta')$ is $\bot$, in which case $\Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash s : \mathbb{c}'$ holds, with $\iota(\mathbb{c})$ and $\mathbb{c}'$ being related with few (possibly none) application of the rule (R-Release-GC): statements $\mathtt{release}\ \beta$ are now typed with (T-Release-Bot) instead of (T-Release).

We can then conclude that $\Gamma' \vdash vm(o, \top, \{l \mid x = v; s\}, q) \quad fut(f, v) : \mathbb{k}'$ with $\mathbb{k}' = vm(o, \top \mid (f', \mathbb{c}', \mathtt{S}),\ \overline{\mathbb{f}}) \parallel f(\mathbb{o}'')$, and behavioral types $\mathbb{k}$ and $\mathbb{k}'$ are related by (R-Read-Fut), and possibly few application of (R-Release-GC).

- Case (Async-Call).

$$\frac{o' = [\![e]\!]_l \quad \overline{v} = [\![\overline{e}]\!]_l \quad f = \text{fresh}(\,)}{vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) \to vm(o, \top, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v})}\ (\text{Async-Call})$$

Let us consider the case when $\Gamma(o') = \top$ (the opposite case $\Gamma(o') = \bot$ is almost identical). By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) : \mathbb{k}$. By reconstruction we have that $\mathbb{k} = vm(o, \top \mid (f', \nu f''\colon \mathtt{m}\ o'(\overline{\mathbb{s}}) \fatsemi \mathbb{c}, \mathtt{S}'),\ \overline{\mathbb{f}})$ and that there is Γ'' such that $\Gamma[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash\ !\mathtt{m}(\overline{e}) : \nu f''\colon \mathtt{m}\ o'(\overline{\mathbb{s}}) \rhd \Gamma''$ and $\Gamma'' \vdash s : \mathbb{c}$.

Let $\iota = [f/f'']$ and lets choose Γ' such that $\Gamma'' = \Gamma'[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathbb{o}']$, by lemma 7.2 we have that $\iota(\Gamma')[(x_i \mapsto \mathbb{x}_i)^{i\in I}][\mathtt{destiny} \mapsto \mathbb{o}'] \vdash vm(o, \top, \{l \mid x = e\,!\mathtt{m}(\overline{e}); s\}, q) \to vm(o, \top, \{l \mid x = f; s\}, q)\ invoc(o', f, \mathtt{m}, \overline{v}) : \mathbb{k}'$ with $\mathbb{k}' = vm(o, \top \mid (f', \mathtt{0} \fatsemi \mathbb{c}', \mathtt{S}'),\ \overline{\mathbb{f}}) \parallel (f'', \mathtt{m}\ o'(\overline{\mathbb{r}}))$, where $\mathbb{c}' = \iota(\mathbb{c})$. Behavioral types $\mathbb{k}$ and $\mathbb{k}'$ are related by (R-Async-Call).

- Case (Release-VM).

$$\frac{o' = [\![e]\!]_l \quad o \neq o'}{vm(o, \top, \{l \mid \mathtt{release}\ e; s\}, q)\ vm(o', a', p', q') \to vm(o, \top, \{l \mid s\}, q)\ \ vm(o', \bot, p', q')}\ (\text{Release-VM})$$

By hypothesis we have that $\Gamma \vdash vm(o, \top, \{l \mid \mathtt{release}\ e; s\}, q)\ vm(o', a', p', q') : \mathbb{k}$ and we can derive that $\mathbb{k} = vm(o, a \mid (f, o'^{\checkmark} \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel vm(o', a' \mid\ \overline{\mathbb{f}'})$. Let $\Gamma' = \Gamma[o' \mapsto \varnothing\bot]$, we have that $\Gamma' \vdash vm(o, \top, \{l \mid s\}, q)\ vm(o', \bot, p', q') : \mathbb{k}'$ with $\mathbb{k}' = vm(o, a \mid (f, \mathbb{c}, \mathtt{S}),\ \overline{\mathbb{f}}) \parallel vm(o', \bot \mid\ \overline{\mathbb{f}'})$. Behavioral types $\mathbb{k}$ and $\mathbb{k}'$ are related by (R-Release-VM)

- Case (RETURN).

$$\frac{v = [\![e]\!]_l \quad f = l(\text{destiny})}{vm(o, \top, \{l \mid \texttt{return}\ e\}, q)\ \ fut(f, \bot) \to vm(o, \top, \{l \mid \varepsilon\}, q)\ \ fut(f, v)}\ (\text{RETURN})$$

  By reconstruction of the proof tree of the hypothesis after the application of rules (BT-PARALLEL), (BT-VM), (BT-PROCESS) and (BT-RETURN), we can derive that $\Bbbk = vm(o, \top \mid (f, [\circledcirc], \mathtt{S}),\ \overline{\mathtt{f}}) \parallel f(\bot)$. Let $\Gamma' = \Gamma$, by hypothesis and by rule (BT-FUT-UNSYNC) we have that $\Gamma' \vdash vm(o, \top, \{l \mid \varepsilon\}, p_2\ \ldots\ p_n)\ \ fut(f, v) : \Bbbk'$ with $\Bbbk' = vm(o, \top \mid (f, \mathtt{0} \rhd \varnothing, \mathtt{S}),\ \overline{\mathtt{f}}) \parallel f(\circledcirc)$. Note that the side condition $\alpha \in \mathtt{R}[{}^{\circledcirc'}/_{\circledcirc}] \Rightarrow \Gamma(\alpha) = \varnothing\bot$ of the rule (BT-FUT-UNSYNC) is a consequence of the constraints $\mathtt{R} = (\mathtt{S} \cup fv(\circledcirc)) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}$ of the (BT-PROCESS) rule. Finally, the behavioral types $\Bbbk$ and $\Bbbk'$ are related by (R-RETURN).

- Case (VM-ERR-RETURN).

$$\frac{f = l(\text{destiny})}{vm(o, \bot, \{l \mid s\}, q)\ fut(f, \bot) \to vm(o, \bot, \{l; \varepsilon\}, q)\ fut(f, err)}\ (\text{VM-ERR-RETURN})$$

  Let $\Gamma \vdash vm(o, \bot, \{l \mid s\}, q)\ fut(f, \bot) : \Bbbk$ where $\Bbbk = vm(o, \bot \mid (f, \mathtt{0} \fatsemi \mathbb{c}, \mathtt{S}),\ \overline{\mathtt{f}}) \parallel f(\bot)$. Let $\Gamma' = \Gamma$, the typing of left and right configurations produces an almost identical derivation tree, hence $\Gamma' \vdash vm(o, \bot, \{l \mid \varepsilon\}, q)\ fut(f, err) : \Bbbk'$, and $\Bbbk' = vm(o, \bot \mid (f, \mathtt{0} \rhd \varnothing, \mathtt{S}),\ \overline{\mathtt{f}}) \parallel fut(f, err)$, which are related by rule (R-RELEASE-BOT).

□

2.7.4 Correctness of the cost computation

The second part of the correctness intends to demonstrate that, given a computation, the costs of configurations therein do not increase. This means that the cost of the first configuration is the greatest one and, therefore, the cost of the program gives an upper bound of the actual cost of every computation of its.

In order to prove this property, we need to extend the notion of cost to runtime types, and show two properties of these cost: i) they are an upper bound for the number of alive VM in the typed configuration; and ii) they decrease when the configuration executes, thus showing that the cost computed for the initial program is an upper bound for the number of alive VM at every step of its execution.

Cost of Runtime Types In this paragraph, we extend the definition of the `translate` function to runtime type $\mathbb{k}$. we first introduce our extension with some useful getters on *well-formed* runtime types (i.e., types corresponding to running configurations)

Definition 7.5. *A runtime type* $\mathbb{k}$ *is* well-formed *iff all the names (object, future, etc) it uses are declared and if it contains the object:* $vm(start, a \mid (f_{start}, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}})$. *Given a well-formed runtime type* $\mathbb{k}$*, we write:* $\mathtt{alive}(\mathbb{k})$ *the number of alive VM in* $\mathbb{k}$ *(i.e., the VM whose state is* $\top$ *in* $\mathbb{k}$*);* $\mathtt{mapping}(\mathbb{k})$ *the mapping giving the state of VM in* $\mathbb{k}$*; and* $\Delta_{\mathbb{k}}$ *the set* $\{ \{o\} \mid o$ *is an object of* $\mathbb{k}\}$ *(i.e.,* $\Delta_{\mathbb{k}}$ *is the identity equivalence relation generated from* $\mathbb{k}$*).*

We extend the function `translate` as presented in Figure 2.9 and define:

Definition 7.6. *The cost equations corresponding to a well-formed runtime type* $\mathbb{k}$*, written* $\mathtt{translate}(\mathbb{k})$ *is defined as follows:*

$$\mathtt{translate}(\mathbb{k}) \stackrel{def}{=} \mathtt{translate}(\Delta_{\mathbb{k}}, \mathtt{mapping}(\mathbb{k}), \mathbb{k})$$

Note in Figure 2.9 that we also need to extend the definition of `translate` on atoms: we need to re-define the cost of a `get` operation as we now have not only the actual futures in the runtime type but also the ones coming from the static typing of the program (the ones that belong to Ψ). Moreover the initial definition of `translate` for the `get` included an extra adjustment derived from the fact that the state of some virtual machine were unknown, which is not the case at runtime.

Cost Solution A *cost solution* Σ is a mapping from cost functions (like $\mathtt{m}^{\Xi}_{\mathrm{net}}(\overline{v})$ or f_{peak}, where f is a future name) to cost expressions. Basically, this cost solution gives cost expressions of the peak and net cost of all methods in the program, as well as, the peak and net cost of all processes in the runtime type. Following [28, Definition 6], we relate cost solutions and runtime types by assuming the existence of a *cost validation predicate* $\models$ between cost solutions and cost equations as generated by the `translate` function in Section 2.6:

Definition 7.7. *Let* C *be a set of cost equations* $\bigcup_{i=1}^{n} f_i(\overline{x_i}) = e_i[se_i]$*; let* Σ *be a function that maps all* $f_i(\overline{x_i})$ *to cost expressions.* Σ *is a* cost solution *of* C*, noted* $\Sigma \models C$*, if for all* $1 \leqslant i \leqslant n$ *and all substitutions* $[\overline{v}/\overline{x_i}]$ *such that* $se_i[\overline{v}/\overline{x_i}]$ *is true, we have that*

$$\Sigma(f_i)(\overline{v}) \geqslant \Sigma(e_i[\overline{v}/\overline{x_i}])$$

where $\Sigma(e)$ *replaces the function calls in* e *by their value in* Σ*.*

Correctness of the Cost Analysis The proof of the correction of our cost analysis is done in four steps. First, we relate the cost equation computed at stating time with the runtime version of the types:

Lemma 7.8. *Let $\overline{M}\ \{\overline{F\ z\ ;}\ s'\}$ be a well-typed program, let $\overline{\mathbb{C}}$, $\mathbb{c}$ be its behavioral type, cn be its initial configuration and $\mathbb{k}$ the runtime type of cn. The for all cost solution Σ with $\Sigma \vDash \texttt{translate}(\overline{\mathbb{C}}, \mathbb{c})$, there exists a cost solution Σ' such that $\Sigma' \vDash \texttt{translate}(\mathbb{k})$ and $\Sigma(\texttt{main}()) = \Sigma'(f_{start\,peak}) + \texttt{alive}(\mathbb{k})$.*

Proof. Let define $\Sigma' = \Sigma[f_{start\,peak} \mapsto \Sigma(\texttt{main}()) - 1]$. By construction (see the construction of the equation page 37 and Figure 2.9), we have that

$$\texttt{translate}(\mathbb{C}_1\ \ldots\ \mathbb{C}_n,\ \mathbb{c}) = \begin{cases} \texttt{translate}(\mathbb{C}_1)\ \cdots\ \texttt{translate}(\mathbb{C}_n) \\ \texttt{main}() = 1 + e_{1,1} \quad [se_1] \\ \vdots \\ \texttt{main}() = 1 + e_{h_1,1} \quad [se_1] \\ \texttt{main}() = 1 + e_{1,2} \quad [se_2] \\ \vdots \\ \texttt{main}() = 1 + e_{h_m,m} \quad [se_m] \end{cases}$$

$$\texttt{translate}(\mathbb{k}) = \begin{cases} f_{start\,peak} = e_{1,1} \quad [se_1] \\ \vdots \\ f_{start\,peak} = e_{h_1,1} \quad [se_1] \\ f_{start\,peak} = e_{1,2} \quad [se_2] \\ \vdots \\ f_{start\,peak} = e_{h_m,m} [se_m] \end{cases}$$

$$\text{with } \texttt{translate}(\varnothing, \varnothing, \alpha, (\texttt{true})\{\,0\,\}, \mathbb{c}) = \textstyle\bigcup_{j=1}^{m} (se_j)\{\,e_{1,j}; \cdots ; e_{h_j,j}\,\}$$

Hence, by construction, we have that $\Sigma' \vDash \texttt{translate}(\mathbb{k})$. Moreover, as $\texttt{alive}(\mathbb{k}) = 1$, we also have that $\Sigma(\texttt{main}()) = \Sigma'(f_{start\,peak}) + \texttt{alive}(\mathbb{k})$. □

Second, we show that any solution of the cost equation generated from a runtime type is an upper bound for the number of live VM in that runtime type:

Lemma 7.9. *Let* $\Gamma \vdash cn : \mathbb{k}$ *and assume there is a cost solution* Σ *such that* $\Sigma \vDash \texttt{translate}(\mathbb{k})$*. Then* $\texttt{alive}(cn) \leqslant \texttt{alive}(\mathbb{k}) + \Sigma(f_{start\,peak})$ *(recall that* $f_{start\,peak}$ *is the peak cost of the main process type of* $\mathbb{k}$*).*

Proof. Observe that $\texttt{alive}(cn) = \texttt{alive}(\mathbb{k})$ by construction of $\mathbb{k}$. We moreover notice that $\Sigma(f_{start\,peak})$ is positive, since peak costs are by construction always positive integers. We thus have the result. □

Third, we show that an upper bound computed by our technique is stable w.r.t. runtime type evolution:

Lemma 7.10. *Suppose given two well-formed runtime types* $\mathbb{k}$ *and* $\mathbb{k}'$ *such that* $\mathbb{k} \rightsquigarrow \mathbb{k}'$*. Then, for all cost solution* Σ *with* $\Sigma \vDash \texttt{translate}(\mathbb{k})$*, there exists* Σ' *such that: i)* $\Sigma' \vDash \texttt{translate}(\mathbb{k}')$*; and ii)* $\texttt{alive}(\mathbb{k}')+\Sigma'(f_{start\,peak}) \leqslant \texttt{alive}(\mathbb{k}) + \Sigma(f_{start\,peak})$*.*

Proof. By Case on the rule used in $\mathbb{k} \rightsquigarrow \mathbb{k}'$. If not stated otherwise, we implicitly use the notation used in the considered reduction rule):

- Cases (R-Skip), (R-Read-Fut-Known), (R-Read-Fut-Unknown), (R-Choice), (R-GC), (R-Return), (R-Async-Call-Err). Because $\texttt{alive}(\mathbb{k}) = \texttt{alive}(\mathbb{k}')$, it is clear that taking $\Sigma' = \Sigma$, we have and $\Sigma(f_{start\,peak}) = \Sigma'(f_{start\,peak})$. Hence, we have the result.

- Case (R-Async-Call). It is easy to remark that $\texttt{translate}(\mathbb{k}')$ differs from $\texttt{translate}(\mathbb{k})$ in the following points: i) the entries $[f'_{\text{peak}} = \texttt{m}^{\Xi}_{\text{peak}}(\overline{v'}); f'_{\text{net}} = \texttt{m}^{\Xi}_{\text{net}}(\overline{v'})]$ (with $\overline{v'} = \Psi(\Xi(\overline{v}))$ and $\Xi = \texttt{EqRel}(\overline{v'})$) have been added to $\texttt{translate}(\mathbb{k}')$; and ii) some references to $\texttt{m}^{\Xi}_{\text{peak}}(\overline{v'})$ and $\texttt{m}^{\Xi}_{\text{net}}(\overline{v'})$ in $\texttt{translate}(\mathbb{k})$ have been replaced in $\texttt{translate}(\mathbb{k}')$ by references to f'_{peak} and f'_{net}. So if we define Σ' as follow, we have the result (as $\texttt{alive}(\mathbb{k}) = \texttt{alive}(\mathbb{k}')$):

$$\Sigma' = \Sigma[f'_{\text{peak}} \mapsto \Sigma(\texttt{m}^{\Xi}_{\text{peak}}(\overline{v'})); f'_{\text{net}} \mapsto \Sigma(\texttt{m}^{\Xi}_{\text{net}}(\overline{v'}))]$$

- Case (R-Bind-Mtd). Given that $\mathbb{c}$ is the behavior of $\texttt{m}\ o(\overline{v})$ and that $\texttt{translate}(\mathbb{k}_1 \parallel \mathbb{k}_2) = \texttt{translate}(\mathbb{k}_1) \cup \texttt{translate}(\mathbb{k}_2)$, $\texttt{translate}(\mathbb{k})$ and $\texttt{translate}(\mathbb{k}')$ produce the same set of equations up to the substitution of formal parameters. By definition of Σ we have the result for $\Sigma' = \Sigma$.

- Case (R-New-VM). Let f_i be all possible branches of f. We define

$$\Sigma' = \Sigma[f_{i,\text{peak}} \mapsto \Sigma(f_{i\_\text{peak}}) - 1][f_{i\_\text{net}} \mapsto \Sigma(f_{i\_\text{net}}) - 1]$$

The definition of Σ' corresponds to the transfer of the cost $+1$ to the set of alive virtual machines. Hence we $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk') - 1$ and $\Sigma'(f_{start\,peak}) = \Sigma(f_{start\,peak}) - 1$, which gives us the result.

- Case (R-RELEASE-VM). If $a' = \top$ we have that $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk')+1$, if $a' = \bot$ we have that $\texttt{alive}(\Bbbk) = \texttt{alive}(\Bbbk')$, thus taking $\Sigma' = \Sigma$, we have the result.

□

2.7.5 Final demonstration

Finally, we can combine all the previous lemma to demonstrate the correctness of Theorem 7.1.

Proof. Let $n = \Sigma(\texttt{main}())$.
The argument proceeds by induction on the number of reduction steps:

- At step 0, we translate the program into its runtime equivalent *cn*. By Lemma 7.8, there exists Σ' such that $\Sigma' \vDash \texttt{translate}(\Bbbk)$ and that we have that $\Sigma'(f_{start\,peak}) + \texttt{alive}(\Bbbk) = n$. Moreover, by Lemma 7.9 we have that $\texttt{alive}(cn) \leqslant \texttt{alive}(\Bbbk) + \Sigma'(f_{start\,peak}) = n$.

- for the inductive case, let Γ, a runtime configuration *cn'*, a runtime type $\Bbbk$ and a cost solution Σ' such that $\Gamma \vdash cn : \Bbbk$, $\Sigma' \vDash \Bbbk$ and $\Sigma'(f_{start\,peak}) + \texttt{alive}(\Bbbk) \leqslant n$ hold. Then by Theorem 7.4, if $cn \to cn'$, there exists Γ' and a runtime type $\Bbbk'$ such that $\Bbbk \rightsquigarrow^* \Bbbk'$ and $\Gamma' \vdash cn' : \Bbbk'$. By Lemma 7.10, there exist a cost solution Σ'' such that $\Sigma'' \vDash \Bbbk'$ and $\texttt{alive}(\Bbbk')+\Sigma''(f_{start\,peak}) \leqslant \texttt{alive}(\Bbbk)+\Sigma'(f_{start\,peak})$. By hypothesis and Lemma 7.9, we that $\texttt{alive}(cn') \leqslant \texttt{alive}(\Bbbk') + \Sigma''(f_{start\,peak}) \leqslant n$.

□

R-Skip
$vm(o, \top \mid (f, \mathtt{0} \fatsemi \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}})$

R-Read-Fut
$vm(o, \top \mid (f, f'^{\checkmark}_{\mathbb{o}} \fatsemi \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \parallel f'(\mathbb{o}') \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}[{}^{\mathbb{o}'}/_{\mathbb{o}}], \mathtt{S}); \overline{\mathbb{f}}) \parallel f'(\mathbb{o}')$

R-Async-Call
$$\frac{f'' \text{ fresh}}{vm(o, \top \mid (f, \nu f' \colon \mathtt{m}\ o'(\overline{v}, \overline{\beta}) \fatsemi \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}[{}^{f''}/_{f'}], \mathtt{S}); \overline{\mathbb{f}}) \parallel (f'', \mathtt{m}\ o'(\overline{v}, \overline{\beta}))}$$

R-Bind-Mtd
$$\frac{\mathtt{m}\ \alpha(\overline{\mathtt{Int}}\,\overline{x}, \overline{\mathtt{VM}}\,\overline{z}) = \mathbb{c},\ \mathbb{o},\ \mathtt{R} \qquad \mathtt{S} = \{o, \overline{o}\}}{vm(o, \top \mid \mathbb{f}; \overline{\mathbb{f}}) \parallel (f, o\ \mathtt{m}(\overline{v}, \overline{o})) \rightsquigarrow vm(o, \top \mid \mathbb{f}; \overline{\mathbb{f}} \cup \{(f, \mathbb{c}[{}^{\overline{v}, \overline{o}}/_{\overline{x}, \overline{z}}][{}^{o}/_{\alpha}], \mathtt{S})\}) \parallel f(\bot)}$$

R-LChoice
$vm(o, \top \mid (f, \mathbb{c} + \mathbb{c}', \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}})$

R-RChoice
$vm(o, \top \mid (f, \mathbb{c} + \mathbb{c}', \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}', \mathtt{S}); \overline{\mathbb{f}})$

R-Ite-T
$$\frac{se = \mathtt{true}}{vm(o, \top \mid (f, (se)\{\,\mathbb{c}\,\}, \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}})}$$

R-Ite-F
$$\frac{se = \mathtt{false}}{vm(o, \top \mid (f, (se)\{\,\mathbb{c}\,\}, \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathtt{0}, \mathtt{S}); \overline{\mathbb{f}})}$$

R-New-VM
$$\frac{o' \text{ fresh}}{vm(o, \top \mid (f, \nu\beta \fatsemi \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}[{}^{o'}/_{\beta}], \mathtt{S}); \overline{\mathbb{f}}) \parallel vm(o', \top \mid \varnothing)}$$

R-Release-VM
$vm(o, \top \mid (f, o'^{\checkmark} \fatsemi \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \parallel vm(o', a' \mid \mathbb{f}'; \overline{\mathbb{f}'}) \rightsquigarrow vm(o, \top \mid (f, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \parallel vm(o', \bot \mid \mathbb{f}'; \overline{\mathbb{f}'})$

R-GC
$vm(o, a \mid (f, \mathtt{0} \rhd \varnothing, \mathtt{S}); \overline{\mathbb{f}} \cup \{\mathbb{f}\}) \rightsquigarrow vm(o, a \mid \mathbb{f}; \overline{\mathbb{f}})$

R-Return
$vm(o, \top \mid (f, [\mathbb{o}], \mathtt{S}); \overline{\mathbb{f}}) \parallel f(\bot) \rightsquigarrow vm(o, \top \mid (f, \mathtt{0} \rhd \varnothing, \mathtt{S}); \overline{\mathbb{f}}) \parallel f(\mathbb{o})$

R-Release-Bot
$vm(o, \bot \mid (f, \mathbb{c}, \mathtt{S}); \overline{\mathbb{f}}) \parallel f(\bot) \rightsquigarrow vm(o, \bot \mid (f, \mathtt{0} \rhd \varnothing, \mathtt{S}); \overline{\mathbb{f}}) \parallel f(err)$

R-Bind-Mtd-Err
$vm(o', \bot \mid \mathbb{f}; \overline{\mathbb{f}}) \parallel (f, o'\ \mathtt{m}(\overline{v})) \rightsquigarrow vm(o', \bot \mid \mathbb{f}; \overline{\mathbb{f}}) \parallel f(err)$

R-Bind-Mtd-Serr
$(f, err\ \mathtt{m}(\overline{v})) \rightsquigarrow f(err)$

R-Release-GC
$vm(o, \top \mid (f, \ldots o'^{\checkmark} \ldots, \mathtt{S}), \overline{\mathbb{f}}) \parallel vm(o', \bot \mid \ldots) \rightsquigarrow vm(o, \top \mid (f, \ldots \mathtt{0} \ldots, \mathtt{S}), \overline{\mathbb{f}}) \parallel vm(o', \bot \mid \ldots)$

R-Context
$$\frac{\mathbb{k} \rightsquigarrow \mathbb{k}'}{\mathbb{k} \parallel \mathbb{k}'' \rightsquigarrow \mathbb{k}' \parallel \mathbb{k}''}$$

Figure 2.8: Operational Semantics of Runtime Types

Extension of `translate` on extended atoms:

$$\texttt{translate}[\Delta,\Psi,\alpha](\overline{e};e)(\mathbb{a}) = \begin{cases} (\Psi,\,\overline{e};e) & \text{when } \mathbb{a} = [o] \\ (\Psi,(\overline{e};e)\sigma;(e)\sigma') & \text{when } \mathbb{a} = f_{\mathbb{o}}{}^{\checkmark} \text{ and } f \notin \mathrm{dom}(\Psi), \text{ where} \\ & \sigma = [f_{\mathrm{peak}}/f] \text{ and } \sigma' = [f_{\mathrm{net}}/f] \\ (\Psi',(\overline{e};e)\sigma;(e)\sigma') & \text{when } \Psi(f) = (\Psi',\mathtt{R},\mathtt{m}\ \beta(\overline{\mathbb{r}},\overline{\beta}) \to \mathbb{o}) \text{ and } \mathbb{a} = f_{\mathbb{o}}{}^{\checkmark} \\ & \text{and } \texttt{EqRel}(\beta,\overline{\beta}) = \Xi \text{ and } \Theta = \Psi' \searrow^{f} \Psi \\ & \text{and } \sigma = [\mathtt{m}^{\Xi}_{\mathrm{peak}}(\overline{\mathbb{r}},\Theta(\Xi(\beta,\overline{\beta})))/_f] \\ & \text{and } \sigma' = [\mathtt{m}^{\Xi}_{\mathrm{net}}(\overline{\mathbb{r}},\Theta(\Xi(\beta,\overline{\beta})))/_f] \end{cases}$$

Extension of `translate` on process types:

$$\texttt{translate}(\Delta,\Psi,o,(f,\mathbb{c},\mathtt{S})) = \begin{cases} f_{\mathrm{peak}} = e_{1,1} & [se_1] \\ \vdots & \\ f_{\mathrm{peak}} = e_{h_n,n} & [se_n] \\ f_{\mathrm{net}} = f_{\mathrm{peak}} & [\alpha = \partial] \\ f_{\mathrm{net}} = e_{h_1,1} & [se_1 \wedge \alpha = \top] \\ \vdots & \\ f_{\mathrm{net}} = e_{h_n,n} & [se_n \wedge \alpha = \top] \end{cases}$$

$$\text{with } \texttt{translate}(\Delta,\Psi[o \mapsto \alpha],(\texttt{true})\,0,\mathbb{c}) = \bigcup_{i=1}^{n}(se_i)\{e_{1,i};\ldots;e_{h_i,i}\}$$

Extension of `translate` on runtime types:

$$\texttt{translate}(\Delta,\Psi,f(\mathbb{o})) = \{f_{\mathrm{peak}} = 0, f_{\mathrm{net}} = 0\}$$

$$\texttt{translate}(\Delta,\Psi,vm(o,\top \mid \mathbb{f}_1;\mathbb{f}_2\ldots,\mathbb{f}_n)) = \bigcup_{i\in 1..n} \texttt{translate}(\Delta,\Psi,o,\mathbb{f}_i)$$

$$\texttt{translate}(\Delta,\Psi,vm(o,\bot \mid \mathbb{f};\overline{\mathbb{f}})) = \varnothing$$

$$\frac{\Xi = \texttt{EqRel}(\overline{v}) \quad \overline{v'} = \Psi(\Xi(\overline{v}))}{\texttt{translate}(\Delta,\Psi,(f,o\ \mathtt{m}(\overline{v}))) = \begin{cases} f_{\mathrm{peak}} = \mathtt{m}^{\Xi}_{\mathrm{peak}}(\overline{v'})) \\ f_{\mathrm{net}} = \mathtt{m}^{\Xi}_{\mathrm{net}}(\overline{v'}) \end{cases}}$$

$$\texttt{translate}(\Delta,\Psi,\mathbb{k} \parallel \mathbb{k}') = \texttt{translate}(\Delta,\Psi,\mathbb{k}) \cup \texttt{translate}(\Delta,\Psi,\mathbb{k}')$$

Figure 2.9: Extension of `translate` to runtime types

2.8 Related Work

The static analysis of resource utilization dates back to the pioneering work by Wegbreit in 1975 [58] that developed a technique for deriving closed-form expressions expressing program upper bounds. After this, a considerable number of cost analysis techniques have been developed. Those ones that are closely related to this contribution are based either on cost equations (solvers) or on amortized analysis.

Techniques based on cost equations

The techniques based on cost equations address cost analysis in three steps: (*i*) extracting relevant information out of the original programs by abstracting data structures to their size and assigning a cost to every program expression, (*ii*) converting the abstract program into cost equations, and (*iii*) solving the cost equations with an automatic tool. Recent advances have been done for improving the accuracy of the calculation upper-bounds for cost equations [2, 3, 4, 5, 6, 7, 8, 9, 20, 28, 37], we refer to [28] for an in-depth comparison of some of these tools. One of the main advantages of these techniques is the treatment of arithmetical expressions and, in particular, Presburger arithmetic conditionals. This gives the support for a more precise cost analysis of non-deterministic statements during the static analysis. As a downside, techniques that extract control flow graphs and use control flow refinement strategies (such as [6, 7]) have a less precise alias analysis and name identity management. These two features are essential for function or procedure abstraction; in fact, a weak approach would jeopardise compositional reasoning when large programs are considered. On the other hand, one of the more important contributions of these approaches has been the possibility to fully target high-end programming languages like `Java` [4, 5, 6] and `C#` [37].

Techniques based on amortized analysis

The techniques based on amortized analysis were introduced by the seminal work of Tarjan [56]. Tarjan's idea was to associate so-called potentials to program expressions by means of type systems (these potentials determine the resources needed for each expression to be evaluated). The connection between the original program and these potentials can be indeed demonstrated by a standard subject-reduction theorem. This technique has received lots of attention by applying it to both functional and imperative languages [13, 23, 43, 44, 45].

The techniques based on types are intrinsically compositional and, more importantly, type derivations can be seen as certificates of abstract descriptions of functions. These methods, in general, do not model the interaction of integer arithmetic with resource usage, which restricts the precision and the domain of analyzable properties. Although, an exception of this is one of the best works based on this approach [41].

There have been few attempts to target high-end programming languages with this technique. That is the case of [13] for `Java` programs and [41] for `C`, although in both cases several restrictions apply to the complex features of both languages.

Our technique aims to combine the advantages of the two type of approaches discussed above. It is modular, like the techniques based on cost equations. It also consists of three steps and it extracts the relevant information of programs by means of a *behavioural* type system, like the technique based on amortized analysis. Therefore, our technique is compositional and can be proved sound by means of a standard subject-reduction theorem. At the same time it is accurate in modelling the interaction of integer arithmetic with resource usage.

A common feature of cost analysis techniques in the literature is that they analyze *cumulative resources*. That is, resources that *do not decrease* during the execution of the programs, such as execution time, number of operations, memory (without an explicit `free` operation). The presence of an *explicit or implicit* release operation considerable entangles the analysis. In [9], a memory cost analysis is proposed for languages with garbage collection. It is worth noticing that the setting of [9] is somewhat less difficult than the one proposed here, because by definition of garbage collection, released memory is always inactive. The impact of the release operation in the cost analysis is thoroughly discussed in [7, 10] by means of the notions of *peak cost* and *net cost* that we have also used in Section 2.6. It is worth noticing that, for cumulative analysis, these two notions coincide while, in non-cumulative analysis (in presence of a release operation), they are different and the *net cost* is key for computing tight upper bounds.

Recently [8] has analysed the cost of a language with explicit releases. We observe that the release operation studied in [8] is used in a very restrictive way: only locally created resources can be released. This constraint guarantees that costs of functions are always not negative, thus permitting the (re)use of non-negative cost models from the cumulative analysis. However, the main difference between our work and the work of [8] is that the target resource in our case are active objects, virtual machines, that may be concurrently hosting operations that, in turn, create and release other virtual

machines.

We conclude by discussing cost analysis techniques for concurrent systems, which are indeed very few [2, 7, 42]. In order to reduce the imprecision of the analysis caused by the nondeterminism, [2, 7] use a clever technique for isolating sequential code from parallel code, called *may-happen-in-parallel* [16]. In contrast, in our work the information of the possibly concurrent operations is carried by the proposed type system, which is a closer idea to what it is proposed in [42]. We notice though, that no one of these contributions considers a concurrent language with a powerful `release` operation that allows one deallocate resources created in outer scopes. In fact, without this operation, one can model the cost by aggregating the sets of operations that can occur in parallel, as in [7], and all the theoretical development is less complex.

2.9 Conclusions

In this Chapter we have presented the `vml` language: a custom modeling language inspired by the Cloud Computing elasticity feature, where resources (virtual machines) can be dynamically allocated and deallocated. This language also includes features from modern programming languages like recursion and a future based concurrency model. Furthermore, resources representing virtual machines can be *active*, carrying processes that may, in turn, acquire and release other machines asynchronously. These are all features that entangle the complexity of the analysis of `vml` programs.

Nevertheless, we have presented a technique for the static analysis of `vml` programs. In particular, we designed a static analysis oriented to the estimation of upper bounds corresponding to the usage of objects representing virtual machines. Our approach is based on a behavioral type system that abstracts the main features of the operations on virtual machines objects. These types are expressive enough to record the effects of complex execution flows like those related to concurrency or recursion.

We demonstrate the soundness of this approach by proving the correctness of the behavioral type system and the translation into cost equations that correctly resemble (an over approximation of) the program execution.

A key detail in this type system is the treatment given to numerical expressions in the management of conditional instructions. This feature may allow achieving a higher precision during the analysis stage when such numerical expressions are expressed in Presburger arithmetics [18]. However, when this is not the case, the cost equations obtained by our technique may be too over approximated. A way to cope with this issue is to include a mech-

anism for adding annotations that can instruct the type system to provide more accurate behaviors [38, 52].

In order to calculate upper bound estimations of the virtual machines usage, the behavioral types of the program are translated into cost equations. The resulting equations can then be analyzed by special solvers oriented to such task. Although, up to date, there have been considered two possible solvers (see Section 3.3), the use of others solvers or even a combination of them could increase the precision of the analysis.

Despite the technical limitations imposed to the language features, we have demonstrated the strength of this approach. We have used the metaphor of virtual machines (VM) usage to illustrate our solution. However it is worth observing that our technique may also be used for the resource analysis of concurrent languages bearing operations of acquire (creation) and release (deletion). For example, our technique, might cover functions such as the classic `malloc` and `free` operations.

Chapter 3

The SRA tool

Summary

We describe the (S)tatic (R)esource (A)nalysis tool [30], which is a prototype implementation based on the theory presented in Chapter 2. We discuss a number of technical details involved in the design of this tool, specially the inference of the behavioral types from Section 2.5, and the transformation of the cost equations in Section 2.6 into an input that is adequate for an off-the-shelf solver – CoFloCo [28]. In this chapter, we also compare the obtained results with those from similar existing tools. Finally we conclude with the discussion of the main remarks about this solution and the future work.

3.1 Introduction

In this chapter we discuss the SRA Tool a prototype implementation of the theory described in Chapter 2. This solution follows the generic approach described in Chapter 1 for the development of a static analysis tool based on behavioral types.

Figure 3.1 particularizes this approach for the case of the VM usage analysis in VML programs. The parsing process follows an straightforward implementation based on an automatic parser generation tool. The typing process, is based on the rules defined by a type system that associates abstract behaviors to vml expressions (see Section 2.5). The type of each program instruction can be automatically inferred by means of a slight modification in the type system rules. Methods side-effects are also calculated as part of this typing process. Both of these tasks are done by means of a fix-point algorithm. The behaviors obtained by the typing process are then trans-

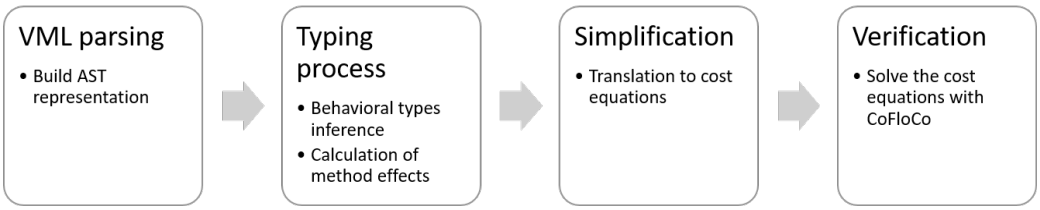

Figure 3.1: Behavioral types based approach for VML virtual machines usage analysis

lated into cost equations. The main idea underneath is to exploit the power of already existing ad-hoc costs equation solvers (CES). In the case of this solution, two CES can be used: PUBS[1] [2] and CoFloCo[2] [28].

Chapter contents

In this Chapter we discuss a prototype implementation of the technique described in Chapter 2. After this introduction, Section 3.2 follows with the description of the behavioral type inference for vml. In Section 3.3 we implement the analysis of the virtual machine usage in vml based on the information abstracted by the type system and with the aid of the CoFloCo solver. An overall example of our technique is discussed in Section 3.4. In Section 3.5 we compare the results of the SRA Tool with those of similar tools in the literature. In Section 3.6 we briefly discuss the SRA Tool demo and the third party tools used in its development. We conclude in Section 3.7 with the discussion of the main remarks about this solution and the future work.

3.2 Type inference

The system defined in Section 2.5 is a type-checking process, meaning that it compels the user to write the behavioural types of methods. In order to relieve users from writing verbose annotations, it is possible to turn the type checking system into a type inference. In fact, in the case of vml, this adaptation is not difficult because the only problematic issue is the inference of method's effects R. This issue is easily solved by changing the rules (T-Method) and (T-Program) in Figure 2.6 into those of Figure 3.2. In particular, rule (Inf-Method) uses a new judgment $\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{T}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\}\ :\ \mathtt{m}\ \alpha(\overline{x}, \overline{\beta})\ \{\mathbb{c}\}\ :\ \mathbb{o},\ \mathtt{R}';\Gamma'$, which also re-

[1] http://costa.ls.fi.upm.es/pubs/pubs.php

[2] https://www.se.tu-darmstadt.de/se/group-members/antonio-flores-montoya/cofloco/

(Inf-Method)

$$\frac{\begin{array}{c}\Gamma(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \circledcirc, \mathtt{R} \qquad \mathtt{S} = \{\alpha\} \cup \overline{\beta} \qquad \circledcirc \notin \mathtt{S} \\ \Gamma[\mathtt{this} \mapsto \alpha][\mathtt{destiny} \mapsto \circledcirc][\overline{x} \mapsto \overline{x}][\overline{z} \mapsto \overline{\beta}][\alpha \mapsto \varnothing\alpha][\overline{\beta} \mapsto \varnothing\overline{\beta}] \vdash_{\mathtt{S}} s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1]\cdots[\mathbb{a}_n \rhd \Gamma_n] \\ \Big(\Gamma_i(\gamma) = \Gamma_j(\gamma)\Big)^{i,j\in 1..n,\ \gamma\in \mathtt{S}\cup fv(\circledcirc)} \qquad \mathtt{R}' = (\mathtt{S} \cup fv(\circledcirc)) \cap \{\gamma \mid \Gamma_1(\gamma) = \mathtt{F}\bot\}\end{array}}{\Gamma \vdash T\ \mathtt{m}\ (\overline{\mathtt{Int}\ x}, \overline{\mathtt{Vm}\ z})\{\overline{F\ y\ ;}\ s\}\ : \mathtt{m}\ \alpha(\overline{x},\overline{\beta})\ \{\mathcal{C}[\mathbb{a}_1 \rhd \varnothing]\cdots[\mathbb{a}_n \rhd \varnothing]\} : \circledcirc,\ \mathtt{R}'; \Gamma[\mathtt{m} \mapsto \alpha(\overline{x},\overline{\beta}) : \circledcirc, \mathtt{R}']}$$

(Inf-Program)

$$\frac{\Gamma \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma \qquad \Gamma[\mathtt{this} \mapsto \mathrm{start}] \vdash_{\mathrm{start}} s : \mathcal{C}[\mathbb{a}_1 \rhd \Gamma_1]\cdots[\mathbb{a}_n \rhd \Gamma_n]}{\Gamma \vdash \overline{M}\ \{\overline{F\ x\ ;}\ s\} : \overline{\mathbb{C}},\ \mathcal{C}[\mathbb{a}_1 \rhd \varnothing]\cdots[\mathbb{a}_n \rhd \varnothing]}$$

Figure 3.2: Behavioural inference rules of method and programs.

turns a new environment Γ' and where $\mathtt{R}'$ may be different from the method's effects stored in $\Gamma(\mathtt{m})$. The main difference between (T-Method) and (Inf-Method) is that the latter one lets Γ' record method's effects computed in the premise. The key constraint in Figure 3.2 is the left premise of (Inf-Program). This premise imposes method definitions to be typed with the *same* input and output environment. That is, the environment Γ is a *fixpoint* of the rule.

In order to compute the fixpoint Γ in the rule (Inf-Program), we notice that the differences between Γ and Γ' in $\Gamma \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma'$, if any, are in methods' effects. In addition, these method's effects are all *finite* because they are always a subset of method's arguments. Therefore, there is a standard technique for computing the *least fixpoint* environment:

1. verify that the transformation $\Gamma \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma'$ is monotone with respect to the subset ordering;

2. let Γ_0 be such that $\Gamma_0(\mathtt{m}) = \alpha(\overline{x},\overline{\beta}) : \circledcirc, \varnothing$ and let $\Gamma_i \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma_{i+1}$. Then there exists a least n such that $\Gamma_n \vdash \overline{M} : \overline{\mathbb{C}}, \Gamma_n$.

It is easy to verify that 1 holds. Henceforth, the correctness of the inference process. For example, let

```
m1 α(β,γ,δ){
   ν f : m2 α(β) ; f✓ ; γ✓
} - , { }
```

```
m2 α(β){
   ν f : m3 α(β) ; f✓
} - , { }
```

```
m3 α(β){
   β✓
} - , { }
```

Then, the reader can verify that the computation of the least fixpoint environment gives the following sequence of environments:

- step 0: $\Gamma_0 = [\texttt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \varnothing,\ \texttt{m2} \mapsto \alpha(\beta) : -, \varnothing,\ \texttt{m3} \mapsto \alpha(\beta) : -, \varnothing]$;
- step 1: $\Gamma_1 = [\texttt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\gamma\},\ \texttt{m2} \mapsto \alpha(\beta) : -, \varnothing,\ \texttt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$;
- step 2: $\Gamma_2 = [\texttt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\gamma\},\ \texttt{m2} \mapsto \alpha(\beta) : -, \{\beta\},\ \texttt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$;
- step 3: $\Gamma_3 = [\texttt{m1} \mapsto \alpha(\beta,\gamma,\delta) : -, \{\beta,\gamma\},\ \texttt{m2} \mapsto \alpha(\beta) : -, \{\beta\},\ \texttt{m3} \mapsto \alpha(\beta) : -, \{\beta\}]$.

3.3 Translation of cost equations

To comply with CoFloCo input formats, we need to encode the VM values and the functions CNEW and CREL into integer values and cost equations, respectively. Let

- $\top$ be encoded by 1, ∂ be encoded by 2, and $\bot$ be encoded by 3. The VM value $\alpha\downarrow$ is encoded by the conditional value $[\alpha = 3]3 + [1 \leqslant \alpha \leqslant 2]2$;
- CNEW and CREL equations are encoded as follows and are fixed for all programs:

```
eq(CNEW(A), 0, [], [A = 3]).
eq(CNEW(A), 1, [], [A < 3]).
```

```
eq(CREL(A), -1, [], [A = 1]).
eq(CREL(A), 0, [], [A > 1]).
```

In order to illustrate the encoding in CoFloCo of the cost equations in Section 2.6, we discuss the equations of the two factorial programs `fact` and `cheap_fact` in Section 2.3. The CoFloCo equations are reported in Figure 3.3.

These equations do not require any comments because they exactly correspond to the output of the `translate` function in Section 2.6. We just observe that, in addition to these equations, one needs to specify a so-called *entry point*, which corresponds to the main body. This entry point has the format

```
entry(METHOD_NAME(LIST_OF_ARGUMENTS):[CONDITIONS])
```

that is instantiated into `entry(main(MAINVM):[])` for the main body. It is also possible to add ad-hoc entries for particular methods, if the user wants to highlight their cost (in particular, our translator adds entries for peak and net costs of every method). For example, the following table reports the outputs (of CoFloCo) corresponding to the entries in the left column.

```
1  eq(fact_peak(A,B), 0, [], [A = 3]).
2  eq(fact_peak(A,B), 0, [], [B = 0]).
3  eq(fact_peak(A,B), 0, [], [B > 0]).
4  eq(fact_peak(A,B), 0, [fact_peak(1,B)], [B > 0]).
5  eq(fact_peak(A,B), 0, [fact_net(1,B)], [B > 0]).
6
7  eq(fact_net(A,B), 0, [], [A = 3]).
8  eq(fact_net(A,B), 0, [fact_peak(A,B)], [A = 2]).
9  eq(fact_net(A,B), 0, [], [A = 1, B = 0]).
10 eq(fact_net(A,B), 0, [fact_net(1, B-1)], [A = 1, B > 0]).
11
12 eq(costly_fact_peak(A,B), 0, [], [A = 3]).
13 eq(costly_fact_peak(A,B), 0, [], [B = 0]).
14 eq(costly_fact_peak(A,B), 0, [], [B > 0]).
15 eq(costly_fact_peak(A,B), 0, [cnew(A)], [B > 0]).
16 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_peak(1, B-1)], [B > 0]).
17 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1)], [B > 0]).
18 eq(costly_fact_peak(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)], [B > 0]).
19
20 eq(costly_fact_net(A,B), 0, [], [A = 3]).
21 eq(costly_fact_net(A,B), 0, [costly_fact_peak(A,B)], [A = 2]).
22 eq(costly_fact_net(A,B), 0, [], [A = 1, B = 0]).
23 eq(costly_fact_net(A,B), 0, [cnew(A), costly_fact_net(1, B-1), crel(1)], [A = 1, B > 0]).
```

Figure 3.3: Cost equations of `fact` and `costly_fact` in `CoFloCo` format

Entry Point	Cost
`entry(fact_net(1,B):[B>=0]).`	`0`
`entry(fact_peak(1,B):[B>=0]).`	`0`
`entry(costly_fact_net(1,B):[B>=0]).`	`0`
`entry(costly_fact_peak(1,B):[B>=0]).`	`max([B,1])`

The *net cost* of `fact` and `costly_fact` is equal to `0` because, in both cases, every created virtual machine is released before the end of the program. For *peak cost*, the number of virtual machines in `fact` is 0. On the other hand, `costly_fact` creates at each step a virtual machine and releases it after the recursive call. This management of virtual machines gives a *peak cost* equal to maximum between `1` and `B`.

3.4 An illustrative example

Consider the following wrong version of the `fake_method` example from Section 2.3, where the programmer has erroneously written "$n + 1$" instead of "$n - 1$" in the recursive invocation (henceforth the recursion is unbounded because the base case is never met). As a result one can expect that the number of created virtual machines is infinite because, at every step, only one of the two created machines is released.

```
Int fake_method(Int n) {
  if (n=0) return 0 ;
  else { VM x, y ; Fut<Int> f, g; Int u, v;
    x = new VM() ; y = new VM() ;
    f = this!double_release(x,x) ;
    u = f.get ;
    g = this!fake_method(n+1) ;
    v = g.get ;
    return 0 ;
  }
}
```

The equations generated by the analysis are shown in Figure 3.4. The results of the analysis of these equations produce a cost of "infinity" for the `main` method, as suspected. However, lets suppose the situation in which (by changing `double_release(x,x)` to `double_release(x,y)`) both VMs are released. In this case the set of equations of `fake_method` remains almost identical with the only difference that the element (`crel(1)` in equations 10, 12, 14 and 18, now appears two times). If we re-run the cost analysis with the new equations we obtain, instead of infinity, a maximum cost of two VMs for the `main` method. Again this is the expected behavior because even though the program runs indefinitely there are no more than two VMs (in addition to the main one) co-existing at the same time.

There is an issue about Figure 3.3 that deserves to be commented. `CoFloCo` outputs an error when it is fed with that set of equations. This is because instructions 8 and 21 are mutually recursive with instructions 5 and 17-18, respectively, and mutual recursion is banned by the analyser. It is worth noticing that 8 and 21 correspond to invocations of `fact_net` and `costly_fact_net` when the carrier is ∂. Intuitively, the idea behind these equations is that when the VM state is unknown the analyzer should assume the worst and consider the highest possible cost (the peak cost) as the net cost. The workaround to solve this problem is to remove the mutual recursion by replacing *in-place* the internal function invocation by its body.

3.5 Related tools and assessments

There are very few tools targeting resource analysis in programs that feature concurrency and negative costs due to explicit release operations. To the best of our knowledge, the most relevant tool in the literature is `SACO`[3], which is based on the results of [7] and is able to address a setting similar to our one. We also compare `SRA` with `C4B`[4] [22], a tool based on amortized analysis

[3]Web site at `http://ei.abs-models.org:8082/clients/web/`

[4]Web site at `http://www.cs.yale.edu/homes/qcar/aaa/`

```
1  eq(fakeMethod01peak(THISVM,N), 0, [], [THISVM = 3]).
2  eq(fakeMethod01peak(THISVM,N), 0, [], [THISVM < 3]).
3
4  eq(fakeMethod01net(THISVM,N), 0, [], [THISVM = 3]).
5  eq(fakeMethod01net(THISVM,N), 0, [], [N = 0, THISVM = 1]).
6
7  eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM)], [N > 0, THISVM < 3]).
8  eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM)], [N > 0, THISVM < 3]).
9  eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),
10                                    doubleRelease011peak(THISVM, 2)], [N > 0, THISVM < 3]).
11 eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM), doubleRelease011net(THISVM, 2),
12                                    crel(1)], [N > 0, THISVM < 3]).
13 eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
14                                    crel(1),fakeMethod01peak(THISVM, N + 1)], [N > 0, THISVM < 3]).
15 eq(fakeMethod01peak(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
16                                    crel(1),fakeMethod01net(THISVM, N + 1)], [N > 0, THISVM < 3]).
17
18 eq(fakeMethod01net(THISVM,N), 0, [], [THISVM = 3]).
19 eq(fakeMethod01net(THISVM,N), 0, [cnew(THISVM),cnew(THISVM),doubleRelease011net(THISVM, 2),
20                                   crel(1),fakeMethod01net(THISVM, N + 1)], [N > 0, THISVM = 1]).
21
22 eq(doubleRelease011peak(THISVM,X), 0, [], [THISVM = 3]).
23 eq(doubleRelease011peak(THISVM,X), 0, [], [THISVM < 3]).
24 eq(doubleRelease011peak(THISVM,X), 0, [crel(X)], [THISVM < 3]).
25 eq(doubleRelease011peak(THISVM,X), 0, [crel(X),crel(3)], [THISVM < 3]).
26
27 eq(doubleRelease011net(THISVM,X), 0, [], [THISVM = 3]).
28 eq(doubleRelease011net(THISVM,X), 0, [crel(X),crel(3)], [THISVM = 1]).
```

Figure 3.4: Cost equations of `fake_method` and `double_release` in `CoFloCo` format

and using the results in [40]. Notwithstanding `C4B` targets only sequential programs, it may be considered as the state-of-the-art of *certified tools* for the automatic static analysis of resource usage bounds.

Some remarks on the tool comparison are in order:

- The three tools target three different languages: `C4B` targets `C`, `SACO` targets `ABS`, and `SRA` targets `vml`. Therefore, in order to compare the three tools, every test program has been rewritten according to the corresponding language of the tool. The sources of these tests are available in github [5].

- Every tool uses different metrics for quantitative analysis. `SRA` quantifies the resource usage by means of the operations `new VM()` and `release`. `C4B` has two metrics: the (*i*) back-edge metrics that assigns a cost of 1 to every back edge in the control-flow graph, thus not allowing negative costs, and (*ii*) the tick metric that uses a special operation

[5] https://github.com/abelunibo/SRA-Tests-Files)

Program	C4B	SACO	SRA
betterThanAmortised	$2 * n$	$2 * n$	$n + m$
greatestCommonDivisor	$Max(n, m)$	*failed*	$n + m$
costlyFactorial	n	n	n
doWorkSync	2	2	2
quicksort	$2 * n$	*not analyzable*	*not analyzable*
SPEED1	$2 * n$	*not analyzable*	*infinite*
doWorkASync	*not analyzable*	n	n
producerConsumer	*not analyzable*	*not analyzable*	3
doubleRelease	*failed*	-2	$[x = y]$-1; $[x \neq y]$-2

Figure 3.5: Comparison of C4B, SACO, and SRA for some simple programs

tick(n) that has a cost of n where n can be either a positive or negative number. SACO, on the other hand, considers both of these kinds of metrics and, in addition, it also support other ones that are specific to the domain of the targeted language.

– Both SACO and SRA separate *peak costs* and *net costs*. This distinction is not made in C4B because it does not address concurrency. Therefore we restrict the results of Table 3.5 only to the *net costs* of the corresponding programs.

– Some programs are *not analyzable* with some of the tools, this is either because *(i)* the programs include some features that are not covered by the analyzer (i.e. concurrency in the case of the C4B tool) or *(ii)* the underlying solver cannot deal with the resulting equations (i.e. non polynomial equations in the case of SRA or non monotonic equations in the case of SACO)

Table 3.5 reports the results of our analysis. It is worth observing that SRA returns better results than both C4B and SACO for the program betterThanAmortized. This program, reported in Figure 3.6, defines a method betterThanAmortised(n,m,x) that recursively invokes itself and, when (n>m) every invocation has cost 1, otherwise it has cost 2. Therefore, the cost of betterThanAmortised(2*n,n,x) is 3*n and this is the result of SRA. However, the other techniques do not recognize that the most costly branch is executed only half of the times and return 2*(2*n)=4*n. Such precision is achieved in SRA by means of the treatment given to numerical expressions, and by the use of a solver (as [28]) capable to deal with conditional branches accurately. This precision is usually lost in techniques based on type systems because of the unifications that are necessary to enforce on the branches of conditionals, in order to type the continuations.

```
Int betterThanAmortized(Int n, Int m, VM x) {
  Fut<Int> f; Int z;
  if (n==0) return 0;
  else if (n>m) {
    VM v = new VM();
    f = x!betterThanAmortized(n-1,m,v);
    z= f.get; return 0; }
  } else {
    VM v = new VM(); VM w = new VM();
    f = v!betterThanAmortized(n-1,m,w);
    z = f.get; return 0;
  }
}

{
  VM x = new VM(); Int z;
  Fut<Int> f = this!betterThanAmortized(2*n, n, x);
  f.get;
}
```

Figure 3.6: The program `betterThanAmortized`

Our behavioral type system, instead, allows the necessary flexibility to overcome this issue. We also observe that `betterThanAmortized` is not an exotic, ad-hoc program but a rather common programming pattern (a sequence of nested conditional statements).

`SRA` also returns remarkable results (*i*) in presence of unbounded recursion – see the example `producerConsumer` –, (*ii*) in cases where the identity of the arguments is relevant for the cost analysis – see the example `doubleRelease` –, and (*iii*) for standard patterns of concurrency – see the examples `doWorkAsync` and `producerConsumer`.

As a main downside of `SRA`, it is unable to compute costs in presence of multi-variate arguments, as in `greatestCommonDivisor` and `SPEED1`. However, it is worth observing that this loss of precision is actually a limitation of the cost equation solver that cannot handle multi-variate arguments in its current version. For a similar reason, `SRA` fails in computing the cost of the `quicksort` program because of a restriction in the `CoFloCo` solver that does not handle non-linear recursion (while `SRA` correctly derives the behavioural types of `quicksort`).

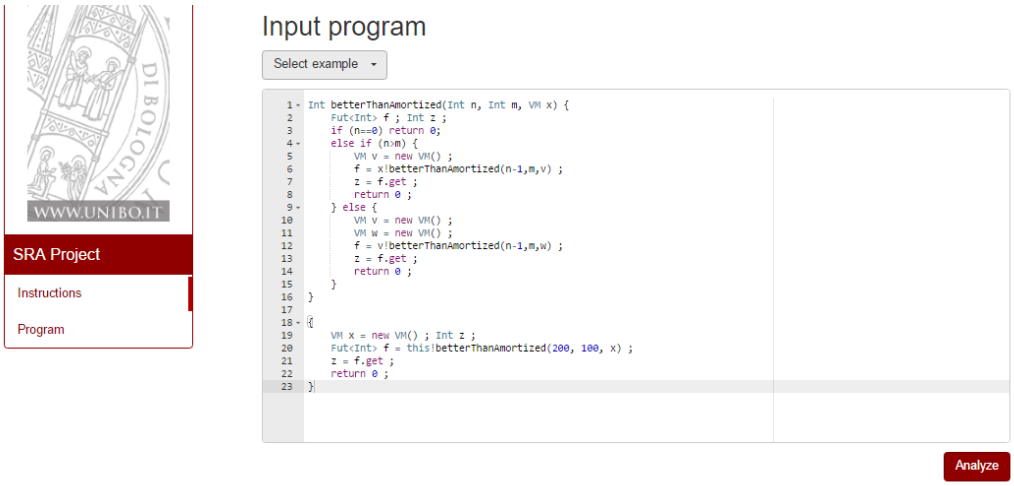

Figure 3.7: SRA tool web demo screen shot

3.6 SRA Deliverable

The SRA tool is available through a dedicated website [30]. The demo page provides a set of default `vml` program examples that are designed to highlight the main features of the tool. The demo page also allows the user to insert custom programs and analyze them.

The prototype components match the general description given in Section 1.2. There are four main components: (i) A `vml` parser, (ii) an inference mechanism, (iii) a finite state representation of the program and (iv) a cost analysis process.

The Figure 3.7 shows the demo page of the `SRA` tool.

Third-party tools involved The orchestration of the `SRA` Tool is implemented in `Java` and it benefits from two out-of-the-box solutions.

The parsing process is done with the help of the `ANTLR` tool[6], this tool provides automatic code generation for taking care of the syntactic analysis of the program as well as the abstract implementation of a semantic checking process that can be later on customized.

There are a couple of solvers capable to solve the cost equations resulting from the translation step in Section 3.3, `PUBS`[7] [2] and `CoFloCo`[8] [28].

[6] http://www.antlr.org/
[7] http://costa.ls.fi.upm.es/pubs/pubs.php
[8] https://www.se.tu-darmstadt.de/se/group-members/antonio-flores-montoya/cofloco/

3.7 Conclusions

To the best of our knowledge the tool here described presents the first static analysis technique that computes upper bounds of virtual machines usage in concurrent programs, that may create and, more importantly, may release such machines.

Our analysis consists of (i) a type system that extracts relevant information about resource usages in programs, called behavioral types; (ii) an automatic translation that transforms these types into cost expressions; and (iii) the application of automatic solvers, like `CoFloCo` or `PUBS`, to compute upper bounds of the usage of virtual machines in the original program.

A relevant property of our technique is its modularity. In this solution we have applied the technique to a small, ad-hoc, language. However, a properly extension of the type system, would allow to apply the analysis to other more complex languages. In addition, by changing the translation algorithm, it is possible to target other solvers that may compute, for certain patterns, better upper bounds. For example, those based on multivariate amortized analysis [40] or in theorem provers [1].

As we have stated there are not currently other static resource analyzers with the same target of the `SRA` Tool. We have, though, assessed the performance of our analyzer by comparing with two state-of-the-art level analyzers: `C4B` and `SACO`. The tested examples demonstrated the programming patterns on which `SRA` excels, like the handling of conditional blocks, the deallocation of resources and the handling of combinations of recursive and concurrent behaviors.

Part II

The deadlock analysis problem

Chapter 4

Deadlock detection in JVML

Summary

This Chapter presents a new solution for the deadlock detection in `Java` programs at static time. This technique targets specifically the `Java` intermediate language – the `Java` bytecode – which has the additional value of allowing the analysis of other languages compiled to the same bytecode, like `Scala`. This technique uses behavioral types to extract abstract models out of the bytecode instructions. These models are subsequently analyzed by means of a fix-point decision algorithm which detects the presence of circular dependencies. The correctness of this technique is demonstrated for a subset of the `Java` bytecode instructions.

4.1 Introduction

In concurrent languages, a *deadlock* is a circular dependency between a set of threads, each one waiting for an event produced by another thread in the set. In the `Java` programming language, deadlocks are usually *resource-related*, namely they are caused by operations ensuring different threads the exclusive access to a set of resources. `Java` programs may also lead to the so-called *communication-related* deadlocks, which are common in network based systems. These deadlocks, which are thoroughly studied in [34, 48], are out of the scope of this work.

As a way to ensure consistency, the Java Language Specification provides different mechanisms to enforce a *should-happen-before* relationship on memory operations over shared variables [36, Chapter 17].

Our analysis focuses on some of these enforcing mechanisms, in particular: the *synchronized constructions*, of both code blocks and method blocks, and

the `Thread.start()` and `Thread.join()` methods. There are also other mechanisms that are not currently handled by the technique here described, such as, the use of `volatile` variables and the higher-level synchronization API defined on package `java.util.concurrent`.

The dependencies enforced by this *should-happen-before* relation may, in some times, produce a circularity. Whenever the execution flow may reach a state with such a circularity, we are in presence of a necessary condition for a deadlocked execution. However, because of the scheduling process this is not a sufficient condition. The Figure 4.1 shows (a time line representation of) some examples of deadlocked programs.

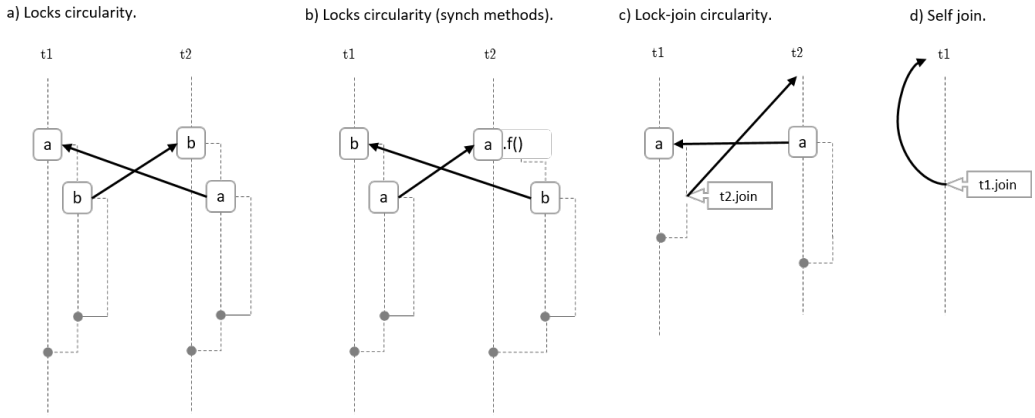

Figure 4.1: Cases of dependency circularities that may lead to deadlocks. (Lock acquisitions are represented with squares, the corresponding release is marked with a circle)

On the other hand, the mere presence of a circularity may be sometimes misleading. This is the case when other *should-happen-before* relations in the program ensure that a state, in which such circularity is verified, will not be reached by the execution flow. See Figure 4.2.

Targeting `JVML`, the `Java` bytecode

The solution here proposed addresses the compilation target of every `Java` application: the `Java` *Virtual Machine Language*, `JVML`, also called `Java` bytecode.

The decision of addressing `JVML` instead of `Java` was motivated by two reasons: `Java` is syntactically too complex and it has no reference semantics. On the contrary, `JVML` a rather simple stack language – it has 198 instructions

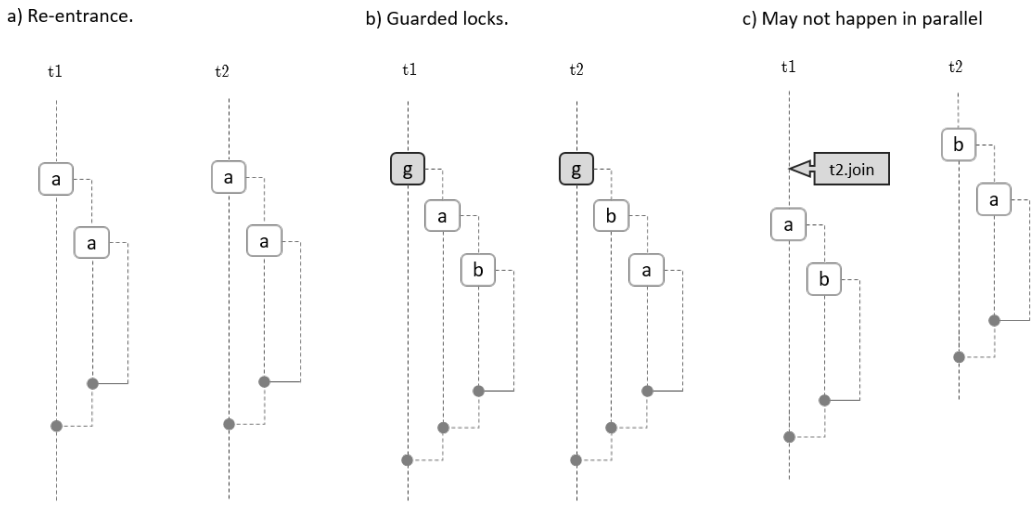

Figure 4.2: Cases of dependency circularities that **do not produce deadlocks**. (Lock acquisitions are represented with squares, the corresponding release is marked with a circle)

– and has a reference semantics that is defined by the behavior of the `Java` *Virtual Machine* (`JVM`) [36, Chapter 6].

This opinion also seems supported by a number of type systems that have been defined in the past for demonstrating the correctness of the `Java` bytecode Verifier [29, 49, 55]. It is worth to observe that reasoning on `Java` bytecode does not narrow our original goal, because as the `Java` compilation target its behavior matches exactly the one intended by the original source. Furthermore, this also does not simplify too much the analysis because it retains the same complex features of `Java`.

Analyzing `JVML` has also other relevant advantages: addressing programming languages that are compiled to the same bytecode, such as `Scala` [53], and the possibility to analyze proprietary software whose sources are not available.

We have defined an inference system that extracts abstract models out of `Java` bytecode instructions. This inference system consists of a number of rules, identical for most of the instructions except for those that have some effect in the synchronization process: invocations, locks acquisitions and releases, object manipulation and control flow operations.

Chapter contents

In this Chapter we discuss the main theoretical components of our deadlock analysis solution. A motivational analysis of the deadlock problems in programming languages and specially in `Java` is presented in Section 4.2. To ease the presentation of the theoretical aspects of this work we propose the language $\texttt{JVML}_d$ in Section 4.3, which is a subset of `JVML` that include basics constructs and instructions for concurrent operations. $\texttt{JVML}_d$ is simpler than `JVML`, it leaves out some complex features of the original language like exception handling, inheritance, arrays manipulation, etc. However, $\texttt{JVML}_d$ itself still poses a difficult challenge for deadlock analysis, with features like recursive invocations or fields manipulation. We present a behavioral type system rules for $\texttt{JVML}_d$ in Section 4.5. The ulterior analysis of these types for the detection of circularities is presented in Section 4.6. We also discuss, in Section 4.7, one more type of deadlock in `Java` and a possible alternative to extend the theory here described in order to detect it. In Section 4.8 we discuss the extensions of the behavioral type system in order to cope with full `JVML` programs. The full proof of the correctness of the presented behavioral type system and its analysis is discussed in Section 4.9. In Section 4.10 we review the existing works in the literature related to the deadlock analysis and to the static analysis of `Java` programs. Finally this chapter concludes with the main remarks about the presented technique.

4.2 Motivation and Problem Statement

Deadlocks are a common threat of concurrent programs, which occur when a set of threads are blocked, due to each attempting to acquire a lock held by another. Such errors are difficult to detect or anticipate, since they may not occur during every execution, and may have catastrophic effects for the overall functionality of the software system. In 1994, a deadlock flaw in the Automated Baggage System of Denver's was one of the causes of losses for more than 100 million dollars[1].

At the time of writing this document, the *Oracle Bug Database*[2] reports more than 40 unresolved bugs due to deadlocks, while the *Apache Issue Tracker*[3] reports around 400 unresolved deadlock bugs. These two databases refer to programs written in `Java`, a mainstream programming language in a lot of domains, such as web and Cloud applications, user applications and

[1] http://ntl.bts.gov/DOCS/rc9535br.html
[2] http://bugs.java.com/
[3] https://issues.apache.org/jira

mobile applications.

The objective of our research is to design and implement a technique capable of detecting potential deadlock bugs of `Java` programs *at compilation time* and, to this aim, we propose an end-to-end automatic static analysis tool. Our implementation handles most of the features of `Java`, including threads, synchronization operations, exceptions, static members, arrays, interfaces, polymorphism, and recursive data types.

Deadlock detection of `Java` programs is a difficult and dubious task. It is difficult because concurrency in `Java` is modeled by threads that may perform read/write operations over shared variables. The order in which concurrent operations are performed depends on the scheduling strategy implemented in the Java Virtual Machine (`JVM`). Therefore deadlocks may not occur during every execution. Additionally, `Java` is a full-fledged programming language, with many features, and with many third-part libraries that are written directly in machine code. It is dubious because `Java` has no reference formal semantics, therefore it is not possible to deliver any soundness proof.

Problem statement

Precise deadlock analysis is, in general, an undecidable problem. This can be proved in the following way (by contradiction):

Undecidability of static deadlock detection. Suppose that for all programs `P` and for all of its possible inputs $\texttt{X}_\texttt{P}$ there is an algorithm `hasDeadlock(P,` $\texttt{X}_\texttt{P}$`)` such that:

`hasDeadlock(P,`$\texttt{X}_\texttt{P}$`)` returns true $\iff$ `P(`$\texttt{X}_\texttt{P}$`)` reaches a deadlocked state

Then consider the following program Z:

```
Z (input_program){
  if(hasDeadlock(Z, input_program))
    return;
  else
    causeDeadlock();
}
```

Consider now running the program `Z` with itself as argument: `Z(Z)`. There are two choices, (i) program `Z(Z)` returns (line 2 is reached), then there is a contradiction because `hasDeadlock(Z,Z)` returns true only if the program deadlocks; and (ii) `Z(Z)` causes a deadlock (line 4 is reached),

then there is also a contradiction because a deadlock is only reached when `hasDeadlock(Z,Z)` returns false. □

In this work we focus on an alternative, but also very challenging, goal:

> Given a program `P` to have an algorithm `mayDeadlock(P)` such that: `mayDeadlock(P)` returns true $\Longrightarrow$ `P` may reach a deadlocked state

Notice that if proved sound an algorithm like `mayDeadlock(P)` may have a great value, since it is able to verify programs as deadlock free. Of course, some precision from this algorithm is required. A naive "return true" implementation is sound, although not useful at all.

This leads also to an implicit problem:

> Given the fact that a program may reach a deadlocked state: to produce the trace that reaches this state.

This information is a key clue for either resolving the deadlock problem or discarding a false positive.

4.3 The language $\texttt{JVML}_d$

In $\texttt{JVML}_d$, a program is a collection of *class files* whose methods have bodies written in $\texttt{JVML}_d$ bytecode. This bytecode is a partial map from *addresses* ADDR to instructions. Addresses, which are ranged over L, L', $\cdots$ are intended to be nonnegative integers and we use the function $L+1$ that returns the least address that is strictly greater than L. When P is a bytecode, we write $dom(P)$ to refer to its domain (the set of addresses) and we assume that $0 \in dom(P)$ for every bytecode P.

Class files use a number of *names*: for classes, ranged over by $\texttt{C}$, $\texttt{D}$, $\cdots$, for fields, ranged over by $\texttt{f}$, $\texttt{f}'$, $\cdots$, for methods, ranged over by $\texttt{m}$, $\texttt{m}'$, $\cdots$, and for local variables, ranged over by x, y, $\cdots$. A possible empty sequence of names or syntactic categories of the following grammar is written by over-lining the name or the syntactic category, respectively. For instance a sequence of local variables is written $\overline{x}$. Class files CF are defined by the grammar:

$$
\begin{array}{lcl}
CF & ::= & class\ \texttt{C}\ \{fields : \overline{FD}\ \ methods : \overline{MD}\} \\
FD & ::= & \texttt{C.f} : \texttt{T} \\
MD & ::= & \texttt{T m}\ (\texttt{C}, \overline{\texttt{T}})\ P \mid \texttt{synchronized T m}\ (\texttt{C}, \overline{\texttt{T}})\ P \\
\texttt{T} & ::= & \top \mid \texttt{int} \mid \texttt{C}
\end{array}
$$

where $\top$ is a special type that include all the other types (any value of any type has also type $\top$). This type will represent values that are unusable in our static semantics. The type name `C` represents a class type. In $\texttt{JVML}_d$, classes are *not recursive*.

4.3.1 Syntax

Instructions *Instr* of $\texttt{JVML}_d$ bytecode are of the following form:

$$
\begin{array}{rcl}
\mathit{Instr} & ::= & \texttt{inc} \mid \texttt{pop} \mid \texttt{push} \mid \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{if}\ L \mid \texttt{goto}\ L \\
& \mid & \texttt{new C} \mid \texttt{putfield C.f} : \texttt{T} \mid \texttt{getfield C.f} : \texttt{T} \\
& \mid & \texttt{invokevirtual C.m}(\overline{\texttt{T}}) \\
& \mid & \texttt{monitorenter} \mid \texttt{monitorexit} \mid \texttt{start C} \mid \texttt{join C} \\
& \mid & \texttt{return}
\end{array}
$$

4.3.2 Informal semantics

The semantics of $\texttt{JVML}_d$ follows exactly the semantics of the corresponding instructions in `JVML` described in [36, Chapter 6]. A full formal description of the operational semantics can be found in Section 4.9.2). The informal meaning of some of these instructions is as follows:

- `inc` increments the content of the stack; `pop` and `push`, respectively, pops and pushes the integer 0 on the stack; `load` x and `store` x respectively loads the value of x on the stack and pops the top value of the stack by storing it in x; `if` L pops the top value of the stack and either jumps to the instruction at address L, if it is nonzero, or goes to the next instruction; `goto` L is the unconditional jump;

- `new C` allocates a new object of type `C`, initializes it and pushes it on top of the stack; `putfield C.f : T` pops the value on the stack and the underlying object value, and assigns the former to the field `f` of the latter; `getfield C.f : T` pops the object on the stack and pushes the value in the field `f` of that object;

- `invokevirtual C.m`$(\texttt{T}_1, \cdots, \texttt{T}_n)$ pops n values from the stack (the arguments of the invocation) and dispatches the method `m` on the object on top of the stack; when the method terminates, the returned value is pushed on the stack;

- `monitorenter`, `monitorexit` are the synchronization primitives that pop the object on the stack and respectively lock and unlock it;

- `start C` creates and starts a new thread for the object on top of the stack; `join C` pops the thread on top of the stack and joins it with the current one. These two operations correspond to `invokevirtual java/lang/Thread/start()` and `invokevirtual java/lang/Thread/join()` on a thread of class `C` in `JVML`, respectively. We separate them from `invokevirtual` in order to provide more structure to our semantics (they have an effect on the set of threads – see the operational semantics in Section 4.9.2);

- `return` terminates program execution.

4.3.3 $\texttt{JVML}_d$ deadlock example: the `Network` class

Figure 4.3 reports a `Java` class called `Network` and some of its bytecode. The corresponding `main` method creates a network of `n` threads by invoking `buildNetwork` – say $t_1, \cdots, t_n$ – that are all potentially running in parallel with the caller – say t_0. Every two adjacent threads share an object, which are also created by `buildNetwork`.

Every thread t_i locks the two adjacent objects, that are passed as (implicit) arguments of the thread, and terminates – this is performed by the method `takeLocks`. It is well-known that, if the network is circular – the thread t_n is sharing one of its objects with t_0 and if all the thread have the strategy of locking objects then a deadlock may occur. On the contrary, if the network is not circular, no deadlock will ever occur.

The `buildNetwork` method will produce a deadlock depending on its actual arguments: it is deadlock-free when it is invoked with two different objects, otherwise it may deadlock (if also $\texttt{n} > 0$). Therefore, in the case of Figure 4.3, the program is deadlocked, while it is deadlock free if we comment the instruction `buildNetwork(n,x,x)` and uncomment `buildNetwork(n,x,y)`.

The problematic issue of `Network` is that the number of threads is not known statically – `n` is an argument of `main`. This is displayed in the bytecode of `buildNetwork` in Figure 4.3 by the instructions at addresses 30 and 37 that respectively created a new thread and start it, and by the recursive invocation at instruction 47.

Our technique is powerful enough to cope with such problems and predict the faulty behavior in case of the invocation `buildNetwork(n,x,x)` and the correct behavior if the invocation is `buildNetwork(n,x,y)`.

```
class Network{

  public void main(int n){
    Object x = new Object();
    Object y = new Object();

    // deadlock
    buildNetwork(n, x, x);

    // no deadlock
    //buildNetwork(n, x, y);
  }

  public void buildNetwork(int n,
  Object x, Object y){
    if (n==0) {

      takeLocks(x,y) ;

    } else {

      final Object z = new Object() ;

      //anonymous Thread child class
      Thread thr = new Thread(){
        public void run(){
          takeLocks(x,z) ;
      }};

      thr.start();
      this.buildNetwork(n-1,z,y) ;
    }
  }

  public void takeLocks(Object x, Object y){
    synchronized (x) {
      synchronized (y) {

      }
    }
  }
}
```

```
public void buildNetwork(int n, Object x, Object y)
0 iload_1              //n
1 ifne 13
4 aload_0              //this
5 aload_2              //x
6 aload_3              //y
7 invokevirtual 24     //takeLocks(x, y):void
10 goto 50
13 new 3
16 dup
17 invokespecial 8     //Object()
20 astore 4            //z
22 new 26
25 dup
26 aload_0             //this
27 aload_2             //x
28 aload 4             //z
30 invokespecial 28    //Network1(this, x, z)
33 astore 5            //thr
35 aload 5             //thr
37 invokevirtual 31    //start():void
40 aload_0             //this
41 iload_1             //n
42 iconst_1
43 isub
44 aload 4             //z
46 aload_3             //y
47 invokevirtual 36    //buildNetwork(n-1, z, y):void
50 return

public void takeLocks(Object x, Object y)
0 aload_1;             //x
1 dup;
2 astore_3;
3 monitorenter;        //acquires x
4 aload_2;             //y
5 dup;
6 monitorenter;        //acquires y
7 monitorexit;         //releases y
8 aload_3;
9 monitorexit;         //releases x
16 return;
```

Figure 4.3: `Java` Network program and corresponding bytecode of methods `buildNetwork` and `takeLocks`. Comments in the bytecode give information of the objects used and/or methods invoked in each instruction

4.4 Preliminaries

Type values and flattened record types. Our technique traces dependencies between objects and it is therefore critical that the static semantics keeps track of object identities. let Obj be the set of *object names*, ranged over by a, b, $\cdots$. Obj includes the two special names *null* and *void* and the *thread names* (in Java, threads are objects as well), which are ranged over by t, t', $\cdots$. We also use x, y, z, $\cdots$ to range over (generic) *names* and X, Y, Z, $\cdots$ to range over *variable names* to be used in the typing of methods and to be instantiated when methods are invoked.

Let $(\mathtt{h}, \leqslant)$ be the lattice of *accesses*, where $\mathtt{h} = \{\_, \mathtt{r}, \mathtt{w}\}$ – with $\_$ meaning no access, $\mathtt{r}$ meaning read access, $\mathtt{w}$ meaning write (and possible read) access – and $\_ \leqslant \mathtt{r} \leqslant \mathtt{w}$. Since $(\mathtt{h}, \leqslant)$ is a lattice, we will use the operation of least upper bound $\sqcup$. Let

– *type values* χ and *flattened record types* ς be the terms

$$\chi \ ::= \ \top \mid \mathtt{int} \mid X \mid a \qquad\qquad \varsigma \ ::= \ (\overline{[\mathtt{f}^{\mathtt{h}} : \chi]}, \mathtt{C})$$

It is worth to remark that flattened record types also display the kind of access to fields and bear their class set, which is a singleton in $\mathtt{JVML}_d$ (but not in full Java, where sets are not singleton because of inheritance, see Section 4.8.5). We always shorten $\mathtt{f}^{\_}$ into $\mathtt{f}$; therefore record types are record types with accesses where the accesses to fields are always $\_$. We use $\mathcal{T}$, $\mathcal{T}'$, $\cdots$ for denoting sets of types $a\overline{[\mathtt{f}^{\mathtt{h}} : \chi]}$.

Let

$$\mathtt{f}^{\mathtt{h}} : \chi \oplus \mathtt{f}^{\mathtt{h}'} : \chi' \stackrel{def}{=} \begin{cases} \mathtt{f}^{\mathtt{h} \sqcup \mathtt{h}'} : \chi & \text{if } \mathtt{h}, \mathtt{h}' \leqslant \mathtt{r} \text{ and } \chi = \chi' \\ \mathtt{f}^{\mathtt{w}} : \mathtt{int} & \text{if } \mathtt{h} \sqcup \mathtt{h}' = \mathtt{w} \text{ and } \chi = \mathtt{int} = \chi' \\ \mathtt{f}^{\mathtt{w}} : \top & \text{otherwise} \end{cases}$$

It is worth to notice that $\mathtt{f}^{\mathtt{h}} : X \oplus \mathtt{f}^{\mathtt{h}'} : Y$ is $\top$ when $X \neq Y$. Let also

$$([\cdots, \mathtt{f}_i^{\mathtt{h}_i} : \chi_i, \cdots], \mathtt{C}) \oplus ([\cdots, \mathtt{f}_i^{\mathtt{h}'_i} : \chi'_i, \cdots], \mathtt{C}) \stackrel{def}{=} ([\cdots, \mathtt{f}_i^{\mathtt{h}_i} : \chi_i \oplus \mathtt{f}_i^{\mathtt{h}'_i} : \chi'_i, \cdots], \mathtt{C})$$

(it is assumed that the fields of the records are the same – because they have the same class type – and the operation is performed on every field).

Environments. The foregoing static semantics uses abstract heaps, called *environments* Γ, which map names to type values or to flattened record types. There are two basic operations on environments: one for *sequential*

composition – the *update* $\Gamma[\Gamma']$ – and one for *parallel composition* – the *merge* $\Gamma \oplus \Gamma'$. They are defined as follows

$$\Gamma[\Gamma'](a) \;=\; \begin{cases} \Gamma(a) & \text{if } a \in dom(\Gamma)\backslash dom(\Gamma') \\ \Gamma'(a) & \text{if } a \in dom(\Gamma')\backslash dom(\Gamma) \\ ([\cdots, \mathtt{f}_i^{\mathtt{h}_i \sqcup \mathtt{h}'_i} : \chi'_i, \cdots], \mathtt{C}) & \text{if } \Gamma(a) = ([\cdots, \mathtt{f}_i^{\mathtt{h}_i} : \chi_i, \cdots], \mathtt{C}) \\ & \text{and } \Gamma'(a) = ([\cdots, \mathtt{f}_i^{\mathtt{h}'_i} : \chi'_i, \cdots], \mathtt{C}) \end{cases}$$

$$(\Gamma \oplus \Gamma')(a) \;=\; \begin{cases} \Gamma(a) & \text{if } a \in dom(\Gamma)\backslash dom(\Gamma') \\ \Gamma'(a) & \text{if } a \in dom(\Gamma')\backslash dom(\Gamma) \\ \Gamma(a) \oplus \Gamma'(a) & \text{otherwise} \end{cases}$$

For example, let $\Gamma = a \mapsto ([\mathtt{f}^{\mathtt{h}} : b], \mathtt{C})$ and $\Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : c], \mathtt{C})$. Then $\Gamma[\Gamma'] = a \mapsto ([\mathtt{f}^{\mathtt{w}} : c], \mathtt{C})$, which is the standard update of an environment plus the recording of the writing operation, while $\Gamma \oplus \Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : \top], \mathtt{C})$, namely the field value is undefined because a *race condition* on the field $\mathtt{f}$ of a caused by the parallel execution of two threads.

In the static semantics, methods have types that are different from type values. This is because, in methods types, the (tree) structure of methods' arguments must be completely described. For this reason and also because class types in $\mathtt{JVML}_d$ are not recursive, we use (unflattened) record types ϑ, which are terms

$$\vartheta \;::=\; \top \mid \mathtt{int} \mid X \mid (a[\overline{\mathtt{f}^{\mathtt{h}} : \vartheta}], \mathtt{C}) \,.$$

Record types that have fields with accesses $\mathtt{f}^{\text{-}}$ (which are always shortened into $\mathtt{f}$) are ranged over by the metavariables τ, τ', $\cdots$. We shorten $a[\,]$ into a; in particular $null[\,]$ and $void[\,]$ are shortened into $null$ and $void$, respectively. Let $root(\cdot)$ be the function defined as follows: $root((void, \mathtt{C})) = \varepsilon$, $root((a[\overline{\mathtt{f}^{\mathtt{h}} : \vartheta}], \mathtt{C})) = a$, $root(\cdot)$ is the identity otherwise. We notice that the sequence $root(void) \cdot S$ is equal to S.

The operation $\oplus$ is extended to record types as follows

$$\vartheta \oplus \vartheta' \;\stackrel{def}{=}\; \begin{cases} (a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta_{\mathtt{f}} \oplus \mathtt{f}^{\mathtt{h}'} : \vartheta'_{\mathtt{f}}, \cdots], \mathtt{C}) & \text{if } \vartheta = (a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta_{\mathtt{f}}, \cdots], \mathtt{C}) \\ & \text{and } \vartheta' = (a[\cdots, \mathtt{f}^{\mathtt{h}'} : \vartheta'_{\mathtt{f}}, \cdots], \mathtt{C}) \\ \top & \text{if } root(\vartheta) \neq root(\vartheta') \end{cases}$$

There is a straightforward way to get an environment from a record type and, conversely, to transform an environment and an object name into its (unflattened) record type. These two functions, called $\mathit{flat}(\cdot)$ and $mk\_tree(\cdot)$, are defined as follows:

$$\mathit{flat}(\vartheta) \;=\; \begin{cases} \varnothing & \text{if } \vartheta \in \{\top, \mathtt{int}, X\} \\ a \mapsto ([\overline{\mathtt{f}^{\mathtt{h}} : root(\vartheta)}], \mathtt{C}) \oplus (\bigoplus_{\vartheta' \in \overline{\vartheta}} \mathit{flat}(\vartheta')) & \text{if } \vartheta = (a[\overline{\mathtt{f}^{\mathtt{h}} : \vartheta}], \mathtt{C}) \end{cases}$$

$$mk\_tree(\Gamma,\chi) \;=\; \begin{cases} \chi & \text{if } \chi \in \{\top, \mathtt{int}, X\} \\ (a[\overline{\mathtt{f} : mk\_tree(\Gamma,\chi)}], \mathtt{C}) & \text{if } \chi = a \text{ and } \Gamma(a) = ([\overline{\mathtt{f^h} : \chi}], \mathtt{C}) \end{cases}$$

We let

$$flat(\{\vartheta_1, \cdots, \vartheta_n\}) \stackrel{def}{=} \bigcup_{1 \leqslant i \leqslant n} flat(\vartheta_i)$$

and

$$mk\_tree(\Gamma, (\chi_1, \cdots, \chi_k)) \stackrel{def}{=} (mk\_tree(\Gamma,\chi_1), \cdots, mk\_tree(\Gamma,\chi_k))$$

we finally define $typeof(\Gamma,\chi) = \mathtt{C}$ if $\Gamma(\chi) = ([\overline{\mathtt{f^h} : \chi'}], \mathtt{C})$, it is $typeof(\Gamma,\chi) = \mathtt{int}$ if $\Gamma(\chi) = \mathtt{int}$, undefined otherwise.

Lams. Behavioral types are *lams* [34], which express object dependencies and method invocations. The syntax of lams, noted ℓ, is the following one:

$$\ell \;::=\; \mathtt{0} \mid (a,b)_t \mid \mathtt{C.m}(a[\overline{\mathtt{f} : \tau}], \overline{\tau}) \to \tau \mid (\nu\, a)\ell \mid \ell \,\&\, \ell \mid \ell + \ell$$

The type $\mathtt{0}$ is the empty type; $(a,b)_t$ specifies a dependency between the object a and the object b that has been created by the thread t; $\mathtt{C.m}(a[\overline{\mathtt{f} : \tau}], \overline{\tau}') \to \tau''$ defines the invocation of $\mathtt{C.m}$ with carrier $a[\overline{\mathtt{f} : \tau}]$, with arguments $\overline{\tau}'$ and with returned record type τ''. The last two elements of the tuple $\overline{\tau}'$ record the thread t that performed the invocation and the last object name b whose lock has been acquired by t. These two arguments will be used by our analyzer to build the right dependencies between callers and callees. The operation $(\nu\, a)\ell$ creates a new name a whose scope is the type ℓ; the operations $\ell \,\&\, \ell'$ and $\ell + \ell'$ define the conjunction and disjunction of the dependencies in ℓ and ℓ', respectively.

The operators $+$ and $\&$ are associative and commutative. As usual, we shorten $\ell_1 + \cdots + \ell_n$ and $\ell_1 \& \cdots \& \ell_n$ into $\sum_{i \in 1..n} \ell_i$ and $\&_{i \in 1..n}\, \ell_i$, respectively.

A *lam program* is a pair $\big(\mathcal{L}, \ell\big)$, where $\mathcal{L}$ is a *finite set* of *function definitions*

$$\mathtt{C.m}(a[\overline{\mathtt{f} : \tau}], \overline{\tau}', t, b) \to \tau'' \;=\; \ell_{\mathtt{C.m}}$$

with $\ell_{\mathtt{C.m}}$ being the *body* of $\mathtt{C.m}$, and ℓ is the *main lam*, the lam corresponding to the method in the role of program entry point. The type τ'' is considered an argument of the lam function as well.

Behavioural Class Table. A *behavioural class table*, noted BCT, is a map from pairs C.m to *method types*:

$$\texttt{C.m} \ : \ (\bar{\tau}, t, a) \rightarrow (\nu\, \bar{c})\langle \vartheta, \mathcal{T}, K, \overline{\vartheta}, \ell \rangle$$

where

- $\bar{\tau}$ is the tuple of record types of the carrier and of the arguments; t is the thread name of the caller and a is the last lock taken (and not released) by the caller;
- $(\nu\, \bar{c})$ are the names that have been created by the method (and occur in the part in angular brackets);
- ϑ is the record type of the returned value; this type may contain fresh object names that are created within the method body;
- $\mathcal{T}$ is the set of thread record types that have been created by the method and that survive (have not been synchronized) after the method execution; (this set only contains record types where root names are fresh);
- K is the set of thread names that occur in the arguments (in $\bar{\tau}$) and that have been synchronized within the body of C.m;
- $\overline{\vartheta}$ is the record type highlighting the accesses and the changes to the arguments performed by the invocation;
- ℓ is the lam of the method C.m.

Let $\text{BCT}(\texttt{C.m}) = (\bar{\tau}, t, a) \rightarrow (\nu\, \bar{c})\langle \vartheta, \mathcal{T}, K, \overline{\vartheta}, \ell \rangle$; we define the *instance of a method type*:

> let σ be a substitution for object names and variable names occurring in $\bar{\tau}, t, a$ such that $(\bar{\tau}, t, a)\sigma = (\overline{\tau'}, t', a')$; let also $\bar{b}$ be a tuple of names such that there is no clash with names in $dom(\sigma)$. Then

$$\text{BCT}(\texttt{C.m})(\overline{\tau'}, t', a')(\bar{b}) \stackrel{def}{=} (\langle \vartheta, \mathcal{T}, K, \overline{\vartheta}, \ell \rangle \{{}^{\bar{b}}/_{\bar{c}}\})\sigma \ .$$

We notice that this instantiation operation requires that $(\bar{\tau}, t, a)\sigma = (\overline{\tau'}, t', a')$, namely actual arguments $\overline{\tau'}$ *must be more specific* than formal parameters patterns $\overline{\tau'}$. For instance, it is not possible that $\overline{\tau'} = \top, \overline{\tau''}$ because the first element of $\bar{\tau}$ is never a $\top$ value (it has at least a root object name).

Typing judgments. Let P be a $\texttt{JVML}_d$ bytecode. P will be typed by the judgment:

$$\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \ell \,;\, \Psi$$

where:

- BCT is the behavioural class table that maps every method of the program to its lam;
- Γ, F, S, Z, T, K are *vectors* indexed by the addresses of P;
- the environment Γ_i maps object names to record types at address i;
- the map F_i takes local variables and returns type values at address i; let $F_\top$ be the map such that $F(x) = \top$, for every x;
- the stack S_i returns a sequence of value types at address i;
- Z_i is the *sequence of object names* locked at address i;
- T_i is the *set of thread names* that are alive at instruction i and that have been created locally either by P or by a method invoked by P;
- K_i returns the *set of thread names* that have been synchronized at instruction i;
- t is the thread name where P is executed;
- ℓ is the behavioral type of P;
- Ψ may be either empty, in this case the judgment is shortened into $\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \ell$, or a tuple $(\vartheta, Z_i, T_i, K_i, \Gamma_i)$. We let $\bigsqcup$ be the commutative and associative operator defined as follows

$$(\vartheta, Z_i, T_i, K_i, \Gamma_i) \bigsqcup \varnothing \stackrel{def}{=} (\vartheta, Z_i, T_i, K_i, \Gamma_i)$$

$$(\vartheta, Z_i, T_i, K_i, \Gamma_i) \bigsqcup (-, Z_i, T_j, K_j, \Gamma_j) \stackrel{def}{=} (\vartheta, Z_i, T_i \cup T_j, K_i \cup K_j, \Gamma_i \oplus \Gamma_j)$$

$$(\vartheta, Z_i, T_i, K_i, \Gamma_i) \bigsqcup (\vartheta', Z_i, T_j, K_j, \Gamma_j) \stackrel{def}{=} (\,\vartheta \oplus \vartheta', Z_i, T_i \cup T_j, K_i \cup K_j, \Gamma_i \oplus \Gamma_j\,)$$

The symbol $-$ represents the absence of a value – this happens in case of abrupt termination, see rules in Figure 4.8. We remark that the above operation is only defined on tuples with the element Z_i equal.

Let $Z_i = a_1 \cdots a_n$ be the sequence of (locked) object names at instruction i – we assume that the leftmost object name is the more recent one that has been taken. We define $\lceil Z_i \rceil$ and $\widehat{Z_i}^t$ as follows

$$\begin{array}{rcl} \lceil Z_i \rceil & = & a_1 \\ \widehat{Z_i}^t & = & \&_{j \in 2..n}(a_j, a_{j-1})_t \end{array}$$

The sequence Z_i may contain twice the same object name; this means that the thread has acquired twice the corresponding lock. For instance $Z_i = a \cdot a$. In this case $\widehat{Z_i}^t = (a, a)_t$, which is not a circular dependency – this is the reason why we index dependencies with thread names. Let $lock_t$ be special object name that is paired with the thread t. The lam $\widehat{T_i}^\Gamma$ is the term

$$\widehat{T_i}^\Gamma = \underset{t \in T_i}{\&} \mathit{typeof}(\Gamma_i, t).\mathtt{run}(\mathit{mk\_tree}(\Gamma, t), t, \mathit{lock}_t)$$

where `run` is the method that is invoked by the `Java` instruction t.`start`(), where t is of type thread.

The function $\mathit{names}(\cdot)$**.** The function $\mathit{names}(i)$ takes an address i and returns a tuple of names whose length depends on the address.

This function is critical when methods are recursive because it allows us to keep the set $\mathcal{T}$ finite. We remind that the type of a method has shape $(\overline{\tau}, t, a) \to (\nu\, \overline{a'})\langle \vartheta, \mathcal{T}, K, \overline{\vartheta}, \ell \rangle$. The bound names $\overline{a'}$ *correspond to (renamings of) a subset of* names returned by $\mathit{names}(\cdot)$ in the method's body – see rules for method definitions in Figure 4.5.

For example, consider a method that starts exactly a new thread t (without joining it) and, in a successive instruction at address i, invokes itself recursively (this happens for instance in method `BuildNetwork` of Figure 4.3). The set $\mathcal{T}$ of this method will be a singleton x and $\mathit{names}(i)$ will return a name t' for instantiating x. If $t \neq t'$, we will get $T_{i+1} = \{t, t'\}$ and we will reach a `return` instruction with this same set. But then, by the rules for method definitions, the unique way to type the recursive method is that $t = t'$, henceforth this enforces $\mathit{names}(i) = t$. (Actually, our solution is more tricky than this because the new thread is a record type ϑ and we require that a same record type already exists in $\mathcal{T}'$.)

4.5 A behavioral type system for $\texttt{JVML}_d$

4.5.1 Typing rules

The type rules corresponding to the behavioral type system for $\texttt{JVML}_d$ programs are reported in Figures 4.4 and 4.5. The most important type rules for the deadlock analysis are commented below.

The rule for `invokevirtual` computes the instance of $\textsc{bct}(\texttt{C.m})$ according to the arguments of the invocation. Let $\langle \vartheta, \mathcal{T}, K', \overline{\vartheta}, \ell \rangle$ be such instance. First of all we notice that the lam ℓ is not used in the rule. This is key for ensuring a compositional typing process, in other words, the type of a method does not depend on the behavior of other methods. Since `invokevirtual` is a sequential invocation, its effects are reported in the environment Γ_{i+1} that types the next instruction, notice the environment update operation. There are several effects: (*i*) the returned value, because it may be an object created by the method; (*ii*) the updates of the arguments of the invocation – the tuple $\overline{\vartheta}$ –; (*iii*) the threads that have been spawned by `C.m` and not synchronized – the set $\mathcal{T}$. We notice that the updates of elements in K' are not reported in Γ_{i+1} because they are already included in (*ii*). Let us comment on the premise $T_{i+1} = (T_i \backslash K') \cup \mathit{root}(\mathcal{T})$. The map T keeps track of the alive threads, whose invocations are reported in the lams – see the term $\widehat{T_i^{\Gamma_i}}$ in the conclusion. It is important to keep the set T_i as small as possible because this impacts on the precision of our analysis. Therefore we remove the threads that have been synchronized in `C.m` – the set K' – and we add those that have been created by the invocation – the set $\mathit{root}(\mathcal{T})$. The lam in the conclusion is the conjunction of the invocation to `C.m`, the invocations in $\widehat{T_i^{\Gamma_i}}$ and the dependencies of the sequence of locks in Z_i – the term $\widehat{Z_i}^t$.

Rules for `start` and `join` are also method invocations: the combination of `start` and `join` on the same thread actually corresponds to the rule for `invokevirtual`. The rule for `start` computes the instance of $\textsc{bct}(\texttt{C.run})$ where `C` is the type of the thread t' on top of the stack S_i. Let ϑ be the effect of this method on t' (this records updates of fields, for example). The environment Γ_{i+1} is therefore the *merge* of ϑ with Γ_i because the new thread t', being in parallel with the current one t, may produce effects at any point in the future. For similar reasons, Γ_{i+1} also contains the threads spawned by `C.run` – the set $\mathcal{T}$. Correspondingly, the set T_{i+1} is augmented with the thread that has been just spawned and those spawned by it. The rule for `join` updates the environment Γ_{i+1} with the whole effects of $\textsc{bct}(\texttt{C.run})$, that is the updates of ϑ and the threads $\mathcal{T}$ that have been spawned by it. The maps T and K are modified in a similar way to `invokevirtual` (the set $\mathcal{T}$ is

Method invocations

$$\frac{\begin{array}{c} P[i] = \texttt{invokevirtual C.m}\ (\texttt{T}_0, \cdots, \texttt{T}_k) \qquad i+1 \in dom(P) \qquad S_i = \chi_k \cdots \chi_0 \cdot S' \\ typeof(\Gamma_i, \chi_0) = \texttt{C} \quad mk\_tree(\Gamma_i, (\chi_0, \cdots, \chi_k)) = (\tau_0, \cdots, \tau_k) \\ \overline{b} = names(i) \quad \text{BCT}(\texttt{C.m})(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil)(\overline{b}) = \langle \vartheta, \mathcal{T}, K', (\vartheta_0, \cdots, \vartheta_k), \ell \rangle \\ \Gamma_{i+1} = \Gamma_i[flat(\vartheta) \oplus (\bigoplus_{i \in 0..k} flat(\vartheta_i)) \oplus flat(\mathcal{T})] \quad S_{i+1} = root(\vartheta) \cdot S' \\ F_{i+1} = F_i \qquad Z_{i+1} = Z_i \qquad K_{i+1} = K_i \cup K' \qquad T_{i+1} = (T_i \backslash K') \cup root(\mathcal{T}) \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t} \ \&\ \texttt{C.m}(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) \to \vartheta}$$

$$\frac{\begin{array}{c} P[i] = \texttt{start C} \qquad i+1 \in dom(P) \qquad S_i = t' \cdot S_{i+1} \\ typeof(\Gamma_i, t') = \texttt{C} \quad mk\_tree(\Gamma_i, t') = \tau \quad \overline{b} = names(i) \\ \text{BCT}(\texttt{C.run})(\tau, t', lock_{t'})(\overline{b}) = \langle void, \mathcal{T}, K', \vartheta, \ell \rangle \\ \Gamma_{i+1} = \Gamma_i \oplus flat(\vartheta) \oplus flat(\mathcal{T}) \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ K_{i+1} = K_i \quad T_{i+1} = T_i \cup \{t'\} \cup root(\mathcal{T}) \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{join C} \qquad i+1 \in dom(P) \qquad S_i = t' \cdot S_{i+1} \\ typeof(\Gamma_i, t') = \texttt{C} \quad mk\_tree(\Gamma_i, t') = \tau \quad \overline{b} = names(i) \\ \text{BCT}(\texttt{C.run})(\tau, t', lock_{t'})(\overline{b}) = \langle void, \mathcal{T}, K', \vartheta, \ell \rangle \quad \Gamma_{i+1} = \Gamma_i \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K'' = K_i \cup K' \quad T'' = T_i \backslash K' \\ K_{i+1}, T_{i+1} = \begin{cases} K'', T'' \backslash \{t'\} & \text{if } t' \in T_i \\ K'' \cup \{t'\}, T'' & \text{if } t' \notin T_i \end{cases} \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t} \ \&\ (\lceil Z_i \rceil, lock_{t'})_t}$$

Locking instructions

$$\frac{\begin{array}{c} P[i] = \texttt{monitorenter} \qquad i+1 \in dom(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ K_i = K_{i+1} \quad T_i = T_{i+1} \quad Z_{i+1} = a \cdot Z_i \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{monitorexit} \qquad i+1 \in dom(P) \\ \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = a \cdot S_{i+1} \\ K_i = K_{i+1} \quad T_i = T_{i+1} \quad Z_{i+1} = Z_i \backslash a \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

Object manipulation instructions

$$\frac{\begin{array}{c} P[i] = \texttt{new C} \qquad i+1 \in dom(P) \quad a = names(i) \quad \mathsf{fields}(\texttt{C}) = \overline{\texttt{f}} \qquad \Gamma_{i+1} = \Gamma_i[a \mapsto (\overline{[\texttt{f} : \top]}, \texttt{C})] \\ F_i = F_{i+1} \quad S_{i+1} = a \cdot S_i \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad T_i = T_{i+1} \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{putfield C.f : T} \qquad i+1 \in dom(P) \\ S_i = \chi \cdot a \cdot S_{i+1} \quad \Gamma_i(a) = [\texttt{f}^{\texttt{h}} : \chi', \overline{\texttt{f}^{\texttt{h}} : \chi'}] \\ \Gamma_{i+1} = \Gamma_i[a \mapsto [\texttt{f}^{\texttt{w}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi'}]] \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad T_i = T_{i+1} \quad K_i = K_{i+1} \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{getfield C.f : T} \qquad i+1 \in dom(P) \\ S_i = a \cdot S' \quad \Gamma_i(a) = [\texttt{f}^{\texttt{h}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi}] \\ \Gamma_{i+1} = \Gamma_i[a \mapsto [\texttt{f}^{\texttt{h} \sqcup \texttt{r}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi}]] \\ S_{i+1} = \chi \cdot S' \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ K_i = K_{i+1} \quad T_i = T_{i+1} \end{array}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \ \&\ \widehat{Z_i^t}}$$

Figure 4.4: Type rules for $\texttt{JVML}_d$ programs – Part I

Stack manipulation instructions

$$\frac{P[i] = \texttt{inc} \qquad i+1 \in dom(P) \quad \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad S_i = \texttt{int} \cdot S' = S_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad T_i = T_{i+1}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

$$\frac{P[i] = \texttt{pop} \qquad i+1 \in dom(P) \quad \Gamma_{i+1} = \Gamma_i \quad F_i = F_{i+1} \quad S_i = \chi \cdot S_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad T_i = T_{i+1}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

$$\frac{P[i] = \texttt{push} \qquad i+1 \in dom(P) \quad \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad \texttt{int} \cdot S_i = S_{i+1} \quad Z_i = Z_{i+1} \quad T_i = T_{i+1} \quad K_i = K_{i+1}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

$$\frac{P[i] = \texttt{load}\ x \qquad i+1 \in dom(P) \quad \Gamma_i = \Gamma_{i+1} \quad F_i = F_{i+1} \quad S_{i+1} = F_i(x) \cdot S_i \quad K_i = K_{i+1} \quad T_i = T_{i+1} \quad Z_i = Z_{i+1}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

$$\frac{P[i] = \texttt{store}\ x \qquad i+1 \in dom(P) \quad \Gamma_i = \Gamma_{i+1} \quad F_{i+1} = F_i[x \mapsto \chi] \quad S_i = \chi \cdot S_{i+1} \quad Z_i = Z_{i+1} \quad T_i = T_{i+1} \quad K_i = K_{i+1}}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

Control flow instructions

$$\frac{P[i] = \texttt{goto}\ L \qquad L \in dom(P) \quad \Gamma_i = \Gamma_L \quad F_i = F_L \quad S_i = S_L \quad K_i = K_L \quad T_i = T_L \quad Z_i = Z_L}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

$$\frac{P[i] = \texttt{if}\ L \qquad i+1, L \in dom(P) \quad \Gamma_{i+1} = \Gamma_i = \Gamma_L \quad F_i = F_{i+1} = F_L \quad S_i = \texttt{int} \cdot S_{i+1} \quad S_{i+1} = S_L \quad K_i = K_{i+1} = K_L \quad T_i = T_{i+1} = T_L \quad Z_i = Z_{i+1} = Z_L}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t}}$$

Return

$$\frac{P[i] = \texttt{return} \qquad S_i = \chi \cdot S'}{\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \widehat{T_i^{\Gamma_i}} \;\&\; \widehat{Z_i^t} \;;\; (mk\_tree(\Gamma_i, \chi), \varnothing, Z_i, T_i, K_i, \Gamma_i)}$$

Method definitions

$$\frac{\begin{array}{c}\text{BCT}(\texttt{C.m}) = (\tau_0, \cdots, \tau_k, t, a) \to (\nu\overline{a'})\langle\vartheta, \mathcal{T}, K', \overline{\vartheta}, \ell\rangle \\ F_1 = F_\top[0 \mapsto root(\tau_0), \cdots, k \mapsto root(\tau_k)] \\ \Gamma_1 = \bigoplus_{i \in 0..k} flat(\tau_i) \\ S_1 = \varepsilon \quad Z_1 = a \quad T_1 = \varnothing \quad K_1 = \varnothing \\ \left(\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \ell_i \,;\, \Psi_i\right)^{i \in dom(P)} \\ \ell = \sum_{i \in dom(P)} \ell_i \quad \bigsqcup_{i \in dom(P)} \Psi_i = (\vartheta, a, \mathcal{T}, K', \Gamma') \\ \overline{\vartheta} = mk\_tree(\Gamma', (root(\tau_0), \cdots, root(\tau_k))) \\ \overline{a'} = fn(\vartheta, \mathcal{T}, K', \overline{\vartheta}) \setminus fn(\tau_0, \cdots, \tau_k, t, a)\end{array}}{\text{BCT} \vdash \texttt{T C.m}\ (\texttt{T}_1, \cdots, \texttt{T}_k)\ P}$$

$$\frac{\begin{array}{c}\text{BCT}(\texttt{C.m}) = (a'[\texttt{f} : \tau], \tau_1, \cdots, \tau_k, t, a) \to (\nu\overline{a''})\langle\vartheta, \mathcal{T}, K', \overline{\vartheta}, \ell\rangle \\ F_1 = F_\top[0 \mapsto a', 1 \mapsto root(\tau_1), \cdots, k \mapsto root(\tau_k)] \\ \Gamma_1 = flat(a'[\texttt{f} : \tau]) \oplus (\bigoplus_{j \in 1..k} flat(\tau_j)) \\ S_1 = \varepsilon \quad Z_1 = a' \cdot a \quad T_1 = \varnothing \quad K_1 = \varnothing \\ \left(\text{BCT}, \Gamma, F, S, Z, T, K, i \vdash_t P : \ell_i \,;\, \Psi_i\right)^{i \in dom(P)} \\ \ell = \sum_{i \in dom(P)} \ell_i \quad \bigsqcup_{i \in dom(P)} \Psi_i = (\vartheta, a' \cdot a, \mathcal{T}, K', \Gamma') \\ \overline{\vartheta} = mk\_tree(\Gamma', (a', root(\tau_1), \cdots, root(\tau_k))) \\ \overline{a''} = fn(\vartheta, \mathcal{T}, K', \overline{\vartheta}) \setminus fn(a'[\texttt{f} : \tau], \tau_1, \cdots, \tau_k, t, a)\end{array}}{\text{BCT} \vdash \texttt{synchronized T C.m}\ (\texttt{T}_0, \cdots, \texttt{T}_k)\ P}$$

Figure 4.5: Type rules for $\texttt{JVML}_d$ programs – Part II

not considered because it has been added in the corresponding `start`. Here we also distinguish the two cases (a) the thread has been spawned locally – $t' \in T_i$ – and (b) the thread has been spawned by the caller – $t' \notin T_i$.

The rules for the locking instructions `monitorenter` and `monitorexit` update the map Z as expected. We notice that this update has an effect on the lam $\widehat{Z_{i+1}}^t$ by augmenting or reducing the dependencies, respectively.

There are two rules for method declarations `C.m`, depending on whether the method is `synchronized` or not. We discuss the latter idiom, the former one being similar. The rule verifies whether what is declared in BCT(`C.m`) does match with what is checked by the judgments of the instructions in its body. Assuming that t is the name of the current thread and a is the last lock it has acquired then we constrain the first instruction of `C.m` body to be typed with F_1 such that the first k local variables are set to the arguments of the invocation and Γ_1 defining such arguments. As regards S_1, T_1 and K_1, they are all empty, while $Z_1 = a$. The matching between BCT(`C.m`) and what is checked by the judgments is performed by collecting the tuples $(\chi, Z_i, T_i, K_i, \Gamma_i)$ of the `return` statements and merging the results. Notice that, if the body of `C.m` does not perform any concurrent operation (`start`, `join`, `monitorenter`, `monitorexit`) then every T_i is empty and every Z_i is a. Therefore, the lam of every instruction is always `0`.

4.5.2 The abstract behavior of the `Network` class

Figure 4.6 details the behavioral type of the `Network` class in Figure 4.3. The types have been simplified for easing the readability: the actual `JaDA` types are more complex and verbose. Comments (in gray) explain the side effects of invocations, other comments (in green) correspond to the lines that are commented in Figure 4.3.

The behavior of `main` begins by calling the constructor of the class `Object`. Notice that, after such invocation, the structure of `x` and `y` is known. Then the type reports the invocation to `buildNetwork`.

The behavior of `takeLocks` is the parallel composition of two dependencies corresponding to the acquisition of the locks of `x` and `y`. Every dependency is formed by the last held lock and the current element. Notice that every method receives an extra argument corresponding to the last acquired lock at the moment of the invocation, in this case that argument is `u`. Dependencies are labeled with the identifier of the current thread, this identifier is also an extra argument, `t` in this case.

The behavior of `buildNetwork` has five states: (*i*) the invocation to `takeLocks`, (*ii*) the creation and initialization of the object `z`, (*iii*) the cre-

```
main(this, t, u) : T{thr} =                          --> returns unsync thread:  thr
  Object.init(x, t, u) + Object.init(y, t, u) +      --> structure of x and y:  x:x[], y:y[]
  //deadlock
  buildNetwork(this, int, x, x, t, u)                --> creates unsync thread:  thr
  //no deadlock
  //buildNetwork(this,int,x,y,t,u)

takeLocks(this, x, y, t, u) = (u, x)_t & (x, y)_t

buildNetwork(this,, x, y|t, u) : T{thr} =            --> returns unsync thread:  thr
  takeLocks(this, x, y, t, u) +
  Object.init(z, t, u) +                             --> z:z[]
  Network$1.init(thr, this, x, z, t, z) +            --> thr:thr[this$0:this[], val$x:x[], val$z:  z[]]
  Network$1.run(thr, thr, u_thr) +
  Network$1.run(thr, thr, u_thr) &                   --> recursive invocation goes in parallel
    buildNetwork(this, int, z, y, t, u)                  with the created thread

Object.init(this, t, u) : this[]= 0                  -->no side effects

Thread.init(this, t, u) : this[] = 0                 -->no side effects

Network$1.init(this, x1, x2, x3, t, u) : this[this$0 : x1, val$x : x2, val$z : x3] =
  Thread.init(this, t, u)

Network$1.run(this[this$0 : x1, val$x : x2, val$z : x3], t, u) =
  takeLocks(x1, x2, x3, t, u)                        --> run methods are treated like any other
```

Figure 4.6: `BuildNetwork`'s lams

ation and initialization of the thread `thr`, (iv) the spawn of `thr`, (v) and the recursive invocation (in parallel with the spawned thread `thr`). The `buildNetwork` method also reports one spawned thread as side effect. This may appear contradictory (because `buildNetwork` spawns n threads). However, in this case `JaDA` is able to detect that `thr` is the only thread (from those created) that may be relevant (for the deadlock analysis) in an outer scope. This deduction is done considering the objects in the record structure of `thr`.

The constructors of `Object` and `Thread` have an empty behavior. On the contrary, the constructor of the class `Network$1` is more complex (`Network$1` is the name the `JVM` automatically gives to the anonymous `Thread` child class[4] instantiated inside the method `buildNetwork` of the class `Network`). Being defined as an inner class, `Network$1` has access to the local variables in the scope in which it has been created, namely the variables `x`, `z` and the `this` reference to the container instance. The `JVM` addresses this by passing these variables to the constructor of the class and assigning them to internal fields, named also with the special character '$' to avoid name collisions: `val$x`,

[4]https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

`val$z` and `this$0`. Notice that the behavior of the constructor keeps track of two important things: the invocation to the constructor of the parent class `Thread.init` and the changes in the carrier object which goes from `this` to `this[this$0:x1, val$x:x2, val$z:x3]` where $\texttt{x}_i$ are the formal arguments.

Finally, the behavior of the `run` method from the class `Network$1` contains only the invocation to the `takeLocks` method. Notice that `run` method assumes a certain structure from the carrier object. We remark that the body of this special method is typed exactly like any other method's body.

4.6 Analysis

Once behavioral types have been computed for the whole $\texttt{JVML}_d$ program, we can analyze the type of the *main* method. The analysis here proposed is based on the algorithm defined in [34, 48] that we briefly overview in this section. This section is complemented by Section 5.3, where it is reported the pseudo-code of the algorithm.

First of all, the semantics of lams is very simple: it amounts to unfolding method invocations. The critical points are that (i) every invocation may create new fresh names and (ii) the method definitions may be recursive. These two points imply that a lam model may be infinite state, which makes any analysis nontrivial. We recall that the states of lams are conjunctions ($\&$) of dependencies and method invocation (because types with disjunctions $+$ are modeled by sets of states with conjunctive dependencies).

The results of [34,48] allow us to reduce to models of lams that are *finite*, *i.e.* finite disjunctions of finite conjunctions of dependencies. In turn, this finiteness makes possible to decide the presence of a so-called *circularity*, namely terms such as $(a, b)_t \,\&\, (b, a)_{t'}$. The reader is referred to [34, 48] for the details about the algorithm.

Actually, dependencies in [34, 48] are not indexed by thread names; here we use more informative dependencies in order to cope with `Java` reentrant locks. In particular $(a, b)_t \,\&\, (b, a)_t$ is not a circularity and, when $t \neq t'$, we carefully separate it from $(a, b)_t \,\&\, (b, a)_{t'}$. The main difference between the algorithm used in our tool and the one in [34,48] is the definition of *transitive closure*, which is the base of the notion of circularity. Let $t \neq t'$ and let $\checkmark$ be a special object name. Let also ℓ be a conjunction $\&$ of dependencies.

- The *transitive closure* of ℓ, noted ℓ^+, is the least conjunction of either dependencies or function invocations that contains ℓ and such that if $(a, b)_t \,\&\, (b, c)_{t'}$ is a sub term of ℓ^+ then either (i) $(a, c)_\checkmark$ is a sub term of ℓ^+, if $t \neq t'$, or (ii) $(a, c)_t$ is a sub term of ℓ^+, if $t = t'$.

- We say that a lam has a *circularity* if it (or a lam equivalent to it when function invocations have been removed) has a sub term ℓ such that $(a, a)_{\checkmark}$ is in ℓ^+.

For example $\ell = (a, b)_t \,\&\, (b, a)_t \,\&\, (b, c)_{t'}$ has no circularity because $\ell^+ = (a, b)_t \,\&\, (b, a)_t \,\&\, (a, a)_t \,\&\, (b, b)_t \,\&\, (b, c)_{t'} \,\&\, (a, c)_{\checkmark}$ does not contain any pair $(a, a)_{\checkmark}$ (this symbol ✓ is a special thread name indicating that the dependency is due to the contributions of two or more threads).

Actually, the notion of transitivity that we use in `JaDA` is a refinement of the one described above. In particular, dependencies are labeled by the sequence of threads that contribute to them. These more informative labels allows us to compute transitive closures of terms like $(a, b)_t \,\&\, (b, c)_t \,\&\, (a, c)_{t'} \,\&\, (c, b)_{t'}$ in a more precise way. In fact, in this case, the mutual exclusion on the initial object a makes the concurrent presence of $(b, c)_t$ and $(c, b)_{t'}$ not possible.

In addition, using sequences of thread names as indexes of dependencies allows us to reconstruct, at least in part, the computation that produce the error. This is crucial for detecting false positives.

4.7 `wait`, `notify` and `notifyAll` methods

This section corresponds to the most recent progress in development of our solution. The current idea is still being analyzed and may not be exempt of mistakes.

We discuss a possible extension to cover the synchronization mechanism enforced by the `wait`, `notify` and `notifyAll` methods. These methods enable programmers to access the `JVM`'s support for expressing thread coordination. They are public and final methods of the class `Object`, so they are inherited by all classes and cannot be modified. The invocations to `wait`, `notify` and `notifyAll` succeed for threads that already hold the lock of the object a on the stack. In this case, the `wait` instruction moves its own thread to the *wait set* of the object a and the object is relinquished by performing as many unlock operations as the integer stored in the lock field of a. The instructions `notify` and `notifyAll` respectively wake up one thread and all the threads in the wait set of o. These threads are re-enabled for thread scheduling, which means competing for acquiring the lock of a. The winner will lock the object a as many unmatched `monitorenter` it did on a before the `wait`-operation.

The *wait-notify* relation between threads can easily lead to deadlocks. There are two possible scenarios:

1. the `o.wait()` operation in t does not *happens-before* a matching `o.notify()` in t';

2. a lock on an object (different than `o`) held by t is blocking the execution of t', thus preventing the invocation of `o.notify()` to happen.

The solution we are investigating uses *lams* with special couples $(a, a^w)_t$ and $(a, a^n)_t$ for representing the wait-notify dependencies. For example, for the wait operation, $(a, a^w)_t$ can be read as "thread t has the lock of `a` and has invoked method `wait` on it". Similarly $(a, a^n)_t$ means "thread t has the lock of `a` and has invoked method `notify` on it". However, these dependencies are not sufficient for deadlock analysis. In fact, the mere presence of a wait couple indicates a potential deadlock. For example a lam like

$$\ell = (a, a^w)_t \ \& \ (a, a^n)_{t'}$$

has to be considered as a deadlock, since nothing ensures that the notification will happen after the wait invocation (in this case it will depend on the scheduler). However, this would be a huge over-approximation. The precision can be enhanced if it is guaranteed that the wait operation *happens-before* the matching notification. For this purpose we keep track of one more type of lam couples: $(a, t')_t$ to be read as "thread t' was spawned by thread t while it was holding the lock on a". Next, consider the following states:

$$\ell_1 = (a, a^w)_t \ \& \ (a, t')_t \ \& \ (a, a^n)_{t'} \qquad \ell_2 = (b, a)_t \ \& \ (a, a^w)_t \ \& \ (b, a)_{t'} \ \& \ (a, a^n)_{t'}$$

We can draw the following conclusions. First, ℓ_1 is not deadlocked, when t' started, t is holding the lock of a, so the notification will not occur until t releases the lock of a, which happens exactly when the `wait` is invoked. Second, ℓ_2 is deadlocked, the situation is similar to the previous one, but after releasing a the thread t still holds the lock on b. The thread t', on the other hand, cannot grab the lock on a until it can grab the lock on b. This implies that the notification is never performed.

Following this analysis, we can define a strategy for deciding whether a wait couple is matched by a notify couple:

> *a lam that contains $(a, a^w)_t$ & $(a, a^n)_{t'}$ is safe whenever (i) t' is spawned by t while holding the lock on* `a` *and (ii) whenever a is the only object in common in the lock chains acquired by t and t' before the wait and the notify operations, respectively.*

We are currently studying the soundness of this solution and we will report our results in a future work.

The bug8012326 As part of the experimentation process of `JaDA` we have analyzed a number of the deadlock-related bugs in the Oracle Bug Database [5]. One of the analyzed errors is the bug number 8012326, described as: "Deadlock occurs when `Charset.availableCharsets()` is called by several threads at the same time". Our analysis revealed that this deadlock happens after the invocation of one native function (that checks for the existing *char sets* in the OS). In particular, one execution path within this function suspends the calling thread while it holds locks on `JVM` objects that are blocking parallel threads. A clever solution would be to manually type the corresponding native function with a wait couple, in this way the deadlock could be predicted with the idea previously described. We remark the flexibility of our technique in allowing simple modifications that can solve new problems.

4.8 Type system extensions

In order to keep the presentation simple, the language $\texttt{JVML}_d$ and the theory developed in Sections 4.5 and 4.6 partially cover `JVML`. In this section we discuss how we address features as static fields, recursive data types, exception handling, inheritance and dynamic dispatch, native method calls, and alternative concurrency models. In particular we will discuss the problems we found and our solutions for extending the basic technique from the previous sections.

4.8.1 Static members

In `Java` it is possible to create fields and methods that belong to the class, rather than to an instance of the class. This is accomplished with the `static` modifier: fields and methods that have the static modifier, called *static fields* and *static methods*, respectively can be accessed from anywhere in the code, without creating an instance of the class. The bytecode retains ad-hoc operations for static members – `putstatic`, `getstatic` and `invokestatic`. In order to adequately deal with static members, we need (i) to extend the arguments in the method types to include the static fields accessed by the method and the corresponding effects, and (ii) to extend the environment Γ_i with definitions of static fields. Point (i) is not problematic, as regards (ii), we admit environments that are defined on class names and return record types containing static fields only. In this way we can address static fields as standard ones – see the first three rules in Figure 4.7.

[5] http://bugs.java.com/

Method invocations

$$\frac{\begin{array}{c}P[i] = \texttt{invokestatic C.m}\ (\texttt{T}_1, \cdots, \texttt{T}_k) \qquad i+1 \in dom(P) \qquad S_i = \chi_k \cdots \chi_1 \cdot S' \\ mk\_tree(\Gamma_i, (\texttt{C}, \chi_1, \cdots, \chi_k)) = (\tau_0, \tau_1, \cdots, \tau_k) \quad \textsc{bct}(\texttt{C.m})(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) = \langle \vartheta, \mathcal{T}', \mathcal{R}', K', (\vartheta_0, \vartheta_1, \cdots, \vartheta_k), \ell \rangle \\ \Gamma_{i+1} = \Gamma_i[flat(\vartheta) \oplus (\bigoplus_{i \in 0..k} flat(\vartheta_i))] \quad S_{i+1} = root(\vartheta) \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \quad K_{i+1} = K_i \cup K' \\ \mathcal{T}_{i+1}, \mathcal{R}_{i+1} = \begin{cases} (\mathcal{T}_i \backslash K') \cup \mathcal{T}', \ \mathcal{R}_i \cup \mathcal{R}' & \text{if } i \text{ is not recursive and } (\mathcal{T}_i \cup \mathcal{R}_i) \cap (\mathcal{T}' \cup \mathcal{R}') = \varnothing \\ \mathcal{T}_i \backslash K', \ (\mathcal{R}_i \backslash K') \cup \mathcal{R}' \cup \mathcal{T}' & \text{otherwise} \end{cases}\end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t} \ \& \ \texttt{C.m}(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) \to \vartheta^{\triangledown}}$$

$$\frac{\begin{array}{c}P[i] = \texttt{putstatic C.f : T} \qquad i+1 \in dom(P) \\ S_i = \chi \cdot S_{i+1} \quad \Gamma_i(\texttt{C}) = [\texttt{f}^{\texttt{h}} : \chi', \overline{\texttt{f}^{\texttt{h}} : \chi'}] \\ \Gamma_{i+1} = \Gamma_i[\texttt{C} \mapsto [\texttt{f}^{\texttt{w}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi'}]] \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c}P[i] = \texttt{getstatic C.f : T} \qquad i+1 \in dom(P) \\ \Gamma_i(\texttt{C}) = [\texttt{f}^{\texttt{h}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi}] \quad \Gamma_{i+1} = \Gamma_i[\texttt{C} \mapsto [\texttt{f}^{\texttt{h} \sqcup \texttt{r}} : \chi, \overline{\texttt{f}^{\texttt{h}} : \chi}]] \\ S_{i+1} = \chi \cdot S_i \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c}P[i] = \texttt{invokestatic C.m}\ (\texttt{T}_1, \cdots, \texttt{T}_k) \quad i+1 \in dom(P) \quad S_i = \chi_k \cdots \chi_1 \cdot S' \quad \texttt{C} \notin dom(\Gamma_i) \\ mk\_tree(\Gamma_i, (\texttt{C}_1, \cdots, \texttt{C}_h)) = (\tau'_1, \cdots, \tau'_h) \quad \textsc{bct}(\texttt{C.<cinit>})(\tau'_1, \cdots, \tau'_h, t, \lceil Z_i \rceil) = \langle \_, \mathcal{T}', \mathcal{R}', K', \vartheta', \vartheta'_1, \cdots, \vartheta'_h, \ell' \rangle \\ \Gamma' = \Gamma_i[flat(\vartheta') \oplus (\bigoplus_{i \in 0..h} flat(\vartheta'_i))] \quad mk\_tree(\Gamma', (\texttt{C}, \chi_1, \cdots, \chi_k)) = (\tau_0, \tau_1, \cdots, \tau_k) \\ \textsc{bct}(\texttt{C.m})(\tau_0, \cdots, \tau_k, root(\vartheta'_1), \cdots, root(\vartheta'_h), t, \lceil Z_i \rceil) = \langle \vartheta, \mathcal{T}'', \mathcal{R}'', K'', (\vartheta_0, \vartheta_1, \cdots, \vartheta_k, \vartheta''_1, \cdots, \vartheta''_h), \ell \rangle \\ \Gamma_{i+1} = \Gamma'[flat(\vartheta) \oplus (\bigoplus_{i \in 0..k} flat(\vartheta_i)) \oplus (\bigoplus_{i \in 1..h} flat(\vartheta''_i))] \\ S_{i+1} = root(\vartheta) \cdot S' \quad F_{i+1} = F_i \quad Z_{i+1} = Z_i \quad K_{i+1} = K_i \cup K' \cup K'' \\ \mathcal{T}_{i+1} = \mathcal{T}_i \backslash (K' \cup K'') \cup \mathcal{T}' \cup \mathcal{T}'' \quad \mathcal{R}_{i+1} = \mathcal{R}_i \backslash (K' \cup K'') \cup \mathcal{R}' \cup \mathcal{R}''\end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t} \ \& \ \Big(\texttt{C.<cinit>}(\tau'_1, \cdots, \tau'_h, t, \lceil Z_i \rceil) \to \_ + \texttt{C.m}(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) \to \vartheta^{\triangledown}\Big)}$$

Figure 4.7: Type rules for static members

There are `Java` static members that may have subtle effects on deadlocks. These are the *static constructors* of classes. These constructors, called `<cinit>` in the bytecode, are executed by means of an implicit mechanism of the `JVM` in correspondence of the first access to the classes, *without any explicit instruction in the bytecode*! To address deadlocks caused by these methods, we assume that classes with static constructors *are not added* to the initial environment. This means that, if the `invokestatic`, `putstatic` and `getstatic` are the first operations accessing to the class `C`, then no-one of the first three rules of Figure 4.7 can be applied because $\Gamma_i(\texttt{C})$ is not defined and, therefore, $mk\_tree(\Gamma_i, \texttt{C})$ will fail. In this case, our system uses alternative rules: Figure 4.7 shows the rule for `invokestatic`. In this rule we are invoking a static method `C.m` in an environment Γ_i that has no binding for `C` – namely the static constructor of `C` has still not been run. This means that the `<cinit>` method of `C` must be type-checked beforehand and the method `C.m` must be analyzed in the resulting environment. We notice

that the method type of C.<cinit> is

$$(\tau'_1, \cdots, \tau'_h, t, b) \to (\nu\ \overline{a})\langle \_, \mathcal{T}', \mathcal{R}', K', \vartheta', \vartheta'_1, \cdots, \vartheta'_h, \ell' \rangle$$

that is, in this case, the carrier of the method – the class C – is not mentioned in the arguments, but the effects of the invocation are reported in the type ϑ'. We notice that the method C.<cinit> has $h + 2$ arguments: the first h documents are classes with static fields that are accessed by this constructor. In the rule of Figure 4.7 we are assuming that the corresponding constructors of $\tau'_1, \cdots, \tau'_h$ have already been analyzed. Otherwise *we use a different type* for C.<cinit> where the arguments contain exactly the classes whose static constructor has been already analyzed – therefore there are 2^h different types of C.<cinit> and, consequently, 2^h different typing rules. The environment resulting from the static constructor is then used to type the invocation of C.m in a similar way to the first rule of Figure 4.7.

An unsound alternative to the case explosion generated by this solution is to assume that all static constructors are executed prior to the main method. This approach is unsound, because it does not actually resembles the semantics of the JVM. However, under the assumption that no concurrent operations take place in these constructors this solution does not affect the correctness of the results.

4.8.2 Exceptions and exception hndlers

When an exception occurs during the execution of a program, the JVM removes from the stack all the elements, except the topmost one (the exception object, which is of a subclass of Throwable), and looks for a handler in the corresponding exception table. The exception table is a sequence of tuples (from, to, target, type), where the first three components are addresses and the last one is the type of the exception. So, if the exception occurred at j, the handler is defined by the first tuple $(i, i', i'', \texttt{type})$ in the exception table such that $i \leqslant j < i'$ and the type of the exception matches type. If a tuple $(i, i', i'', \texttt{type})$ is found, the control is given to the instruction at i''; otherwise the exception is re-thrown to the caller. If no handler is found for that exception, the program terminates (in an abrupt way).

In order to model exceptions at static time, we assume that method bodies P also include an exception table that is accessed by means of the notation P.ET. The table P.ET takes in input a pair $(i, \texttt{C})$, namely the instruction index i and the type C of the exception, and returns an instruction index k.

Because a method may return an exception object, we need to extend the behavioral class table, in order to cope with abrupt terminations. Therefore

BCT maps $\mathtt{C.m}(\bar{\tau}, t, a)$ to a tuple

$$(\nu\, \overline{a'})\langle \vartheta, \overline{\vartheta_{\mathtt{e}}}, \mathcal{T}', \mathcal{R}', K', \overline{\vartheta}, \ell \rangle$$

where $\overline{\vartheta_{\mathtt{e}}}$ is a finite sequence of record types that is linear with respect to the classes in sequence (pairwise different types have pairwise different subclasses of `Throwable`) that is used to select the right handler in the exception table. In case of abrupt termination, see rules in Figure 4.8 – the returned value ϑ is absent. We model the absence of ϑ with the symbol $-$. This requires an extension of the $\bigsqcup$ operation in Section 4.6 for tuples $(\vartheta, Z_i, \mathcal{T}_i, \mathcal{R}_i, K_i, \Gamma_i)$:

$$(\vartheta, Z_i, \mathcal{T}_i, \mathcal{R}_i, K_i, \Gamma_i) \bigsqcup (-, Z_i, \mathcal{T}_j, \mathcal{R}_j, K_j, \Gamma_j) \stackrel{def}{=} (\vartheta, Z_i, \mathcal{T}_i \cup \mathcal{T}_j, \mathcal{R}_i \cup \mathcal{R}_j, K_i \cup K_j, \Gamma_i \oplus \Gamma_j)$$

The rule for `invokevirtual` in presence of exceptions is defined in Figure 4.8. The first six lines of this rule are similar to those of Figure 4.4, except for the return tuple of the invoked method that also contains $\overline{\vartheta_{\mathtt{e}}}$. This tuple of record types is used to select the right handler according to the definitions in $P.\mathtt{ET}$.

In particular, for every $(a[\overline{\mathtt{f^h} : \vartheta'}], \{\mathtt{C_e}\}) \in \overline{\vartheta_{\mathtt{e}}}$, we select the index $P.\mathtt{ET}(i, \mathtt{C_e})$, if any. Let this index be h; we type the instruction at h by constraining S_h to be a single element stack and K_h and $\mathcal{T}_h$ to be over-approximations of those used for normal termination. Then K_h is taken to be equal to K_i – in case of abrupt termination we assume that no thread in input is joined by the called method – and, correspondingly, $\mathcal{T}_h$ is $\mathcal{T}_i \cup \mathit{root}(\mathcal{T})$ – we assume that every thread created in case of normal termination has been already created. If the exception is not handled by the exception table then we report all the informations in Ψ in order to be lifted to the method type of P. Figure 4.8 also defines the rule for the static correctness of `athrow`. This rule constraints the state of the initial instruction of the corresponding handler, if any, otherwise it stores the record and its type in Ψ, in order to be returned to the caller.

Method invocations with exceptions and athrow

$$\frac{\begin{array}{c} P[i] = \texttt{invokevirtual C.m}\ (\texttt{T}_0, \cdots, \texttt{T}_k) \qquad i+1 \in dom(P) \qquad S_i = \chi_k \cdots \chi_0 \cdot S' \\ typeof(\Gamma_i, \chi_0) = \texttt{C} \quad mk\_tree(\Gamma_i, (\chi_0, \cdots, \chi_k)) = (\tau_0, \cdots, \tau_k) \\ \textsc{bct}(\texttt{C.m})(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) = \langle \vartheta, \overline{\vartheta_{\texttt{e}}}, \mathcal{T}', \mathcal{R}', K', (\vartheta_0, \cdots, \vartheta_k), \ell \rangle \\ \Gamma_{i+1} = \Gamma_i[flat(\vartheta) \oplus (\bigoplus_{i \in 0..k} flat(\vartheta_i))] \quad S_{i+1} = root(\vartheta) \cdot S' \quad F_{i+1} = F_i \\ Z_{i+1} = Z_i \qquad K_{i+1} = K_i \cup K' \\ \mathcal{T}_{i+1}, \mathcal{R}_{i+1} = \begin{cases} (\mathcal{T}_i \backslash K') \cup \mathcal{T}',\ \mathcal{R}_i \cup \mathcal{R}' & \text{if } i \text{ is not recursive} \\ & \text{and } (\mathcal{T}_i \cup \mathcal{R}_i) \cap (\mathcal{T}' \cup K') = \varnothing \\ \mathcal{T}_i \backslash K',\ (\mathcal{R}_i \backslash K') \cup \mathcal{T}' \cup \mathcal{R}' & \text{otherwise} \end{cases} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i}\ \&\ \widetilde{\mathcal{R}_i}\ \&\ \widehat{Z_i^t}\ \&\ \texttt{C.m}(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) \to \vartheta^{\nabla}\ ,\ \bigsqcup_{\texttt{C}_{\texttt{e}} \in typeof(\overline{\vartheta_{\texttt{e}}})} \Psi_{\texttt{C}_{\texttt{e}}}}$$

$$\left(\begin{array}{l} P.\texttt{ET}(i, \texttt{C}_{\texttt{e}}) = h \\ \quad \text{implies} \quad \begin{array}{l} \Gamma_h = \Gamma_{i+1}[flat((a[\overline{\texttt{f}^{\texttt{h}} : \vartheta'}], \{\texttt{C}_{\texttt{e}}\}))] \quad S_h = a \\ F_h = F_i \quad Z_h = Z_i \quad K_h = K_i \quad \Psi_{\texttt{C}_{\texttt{e}}} = \varnothing \\ \mathcal{T}_h, \mathcal{R}_h = \begin{cases} \mathcal{T}_i \cup \mathcal{T}',\ \mathcal{R}_i \cup \mathcal{R}' & \text{if } i \text{ is not recursive} \\ & \text{and } (\mathcal{T}_i \cup \mathcal{R}_i) \cap (\mathcal{T}' \cup K') = \varnothing \\ \mathcal{T}_i,\ \mathcal{R}_i \cup \mathcal{T}' \cup \mathcal{R}' & \text{otherwise} \end{cases} \end{array} \\ \\ (i, \texttt{C}_{\texttt{e}}) \notin dom(P.\texttt{ET}) \\ \quad \text{implies} \quad \begin{array}{l} \Gamma' = \Gamma_i[flat((a[\overline{\texttt{f}^{\texttt{h}} : \vartheta'}], \{\texttt{C}_{\texttt{e}}\}))] \\ \Psi_{\texttt{C}_{\texttt{e}}} = \begin{cases} (-, [\texttt{C}_{\texttt{e}} \mapsto a], Z_i, \mathcal{T}_i \cup \mathcal{T}', \mathcal{R} \cup \mathcal{R}', K_i, \Gamma') & \text{if } i \text{ is not recursive} \\ & \text{and } (\mathcal{T}_i \cup \mathcal{R}_i) \cap (\mathcal{T}' \cup K') = \varnothing \\ (-, [\texttt{C}_{\texttt{e}} \mapsto a], Z_i, \mathcal{T}_i, \mathcal{R} \cup \mathcal{T}' \cup \mathcal{R}', K_i, \Gamma') & \text{otherwise} \end{cases} \end{array} \end{array} \right)^{(a[\overline{\texttt{f}^{\texttt{h}}:\vartheta'}], \{\texttt{C}_{\texttt{e}}\}) \in \overline{\vartheta_{\texttt{e}}}}$$

$$\frac{P[i] = \texttt{athrow} \quad S_i = \chi \cdot S' \quad typeof(\Gamma_i, \chi) = \texttt{C}}{\left(\begin{array}{lll} P.\texttt{ET}(i, \texttt{C}) = h & \text{implies} & \Gamma_h = \Gamma_i \quad S_h = \chi \quad F_h = F_i \quad Z_h = Z_i \\ & & K_h = K_i \quad \mathcal{T}_h = \mathcal{T}_i \quad \mathcal{R}_h = \mathcal{R}_i \quad \Psi = \varnothing \\ \\ (i, \texttt{C}) \notin dom(P.\texttt{ET}) & \text{implies} & \Psi = (-, [\texttt{C} \mapsto \chi], Z_i, \mathcal{T}_i, \mathcal{R}_i, K_i, \Gamma_i) \end{array} \right)}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i}\ \&\ \widetilde{\mathcal{R}_i}\ \&\ \widehat{Z_i^t}\ ,\ \Psi}$$

Figure 4.8: The type rules for for managing exceptions

4.8.3 Arrays

In `Java`, arrays are objects, are dynamically created, and may be assigned to variables of type `Object`. In `JVML` there are two instructions for creating single-dimension arrays `newarray`, for arrays containing primitive data types, and `anewarray`, for arrays containing objects. (The operation `multinewarray` for creating multi-dimensional arrays is handled by our prototype in a similar way to the previous ones.) In this section we focus on single-dimensional arrays containing objects because the other ones are not problematic for deadlock analysis.

The typical pattern for creating arrays objects in `Java`, where the elements store pairwise different new objects (of class `Object`) is

```
L1: bipush 10
    anewarray java/lang/Object
    astore 1
L2: iconst_0
    istore 2
L3: iload 2
    bipush 9
    if_icmpgt L4
    aload 1
    iload 2
    new java/lang/Object
    dup
    invokespecial java/lang/Object.<init> ()
    aastore
    iload 2
    iinc
    istore 2
    goto L3
L4: return
```

Namely, an array of ten elements is created and the store operation initializing the element is performed in a cycle starting at `L3`.

Our technique does not identify integers; henceforth it is not possible to distinguish different indexes and the elements stored in the corresponding locations. Therefore, arrays are over-approximated by assuming that they *store the same element at every index*. Because of this, we are able to analyze those cases where the elements of the array are "compatible" (see below). For instance, our analysis covers arrays that are initialized with *pairwise different new objects*.

Arrays of class `C` are modeled by structures retaining single values, namely records $([\texttt{idx} : \chi], [\texttt{C}])$. Let $\texttt{Env}(\chi, \texttt{C}, \overline{b_\$})$ be an environment defining χ of type `C` and, whenever the value is a record, every name occurring in the fields of χ, and so on recursively. That is, the function $\texttt{Env}(\chi, \texttt{C}, \overline{b_\$})$ returns a *representative* for arrays' values that stores in the possible fields either $\top$ or `int` or a variable or an object name already defines in Γ_i (this environment is left implicit in the notation) or a name that is subscribed with \$. (Names indexed by \$ are managed in an ad-hoc way by the analyzer, see below.)

Let `Eq` be a mapping from names $a_\$$ to names. Let $\chi' \rhd^{\texttt{Eq}}_{\Gamma_i} \chi$, say χ' *is* `Eq`*-compatible with* χ, if one of the following holds:

1. $\chi' = \chi$,

Array manipulation instructions

$$\frac{\begin{array}{c}P[i] = \texttt{anewarray C} \qquad i+1 \in dom(P) \\ (a, \overline{b_\$}) = names(i) \qquad \Gamma_{i+1} = \Gamma_i[a \mapsto ([\texttt{idx}^{\texttt{w}} : a_\$], [\texttt{C}])] + \texttt{Env}(a_\$, \texttt{C}, \overline{b_\$}) \\ F_i = F_{i+1} \quad S_{i+1} = a \cdot S_i \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i}^t}$$

$$\frac{\begin{array}{c}P[i] = \texttt{aastore} \qquad i+1 \in dom(P) \\ S_i = \chi \cdot \texttt{int} \cdot a \cdot S_{i+1} \quad \text{there is } \texttt{Eq}\text{: } \chi \rhd^{\texttt{Eq}}_{\Gamma_i} a_\$ \\ \Gamma_{i+1} = \Gamma_i[a \mapsto ([\texttt{idx}^{\texttt{w}} : a_\$], [\texttt{C}])][\chi \Rightarrow_{\Gamma_i} a_\$] \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \\ \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i}^t}$$

$$\frac{\begin{array}{c}P[i] = \texttt{aaload} \qquad i+1 \in dom(P) \\ S_i = \texttt{int} \cdot a \cdot S' \quad \Gamma_i(a) = ([\texttt{idx}^{\texttt{h}} : a_\$], [\texttt{C}]) \\ \Gamma_{i+1} = \Gamma_i[a \mapsto ([\texttt{idx}^{\texttt{h} \sqcup \texttt{r}} : a_\$], [\texttt{C}])] \\ S_{i+1} = a_\$ \cdot S' \quad F_i = F_{i+1} \quad Z_i = Z_{i+1} \\ K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i}^t}$$

$$\frac{\begin{array}{c}P[i] = \texttt{putfield C.f : T} \qquad i+1 \in dom(P) \\ S_i = \chi \cdot a_\$ \cdot S_{i+1} \quad \Gamma_i(a_\$) = ([\texttt{f}^{\texttt{h}} : \chi', \overline{\texttt{f}^{\texttt{h}} : \chi'}], \texttt{C}) \quad \text{there is } \texttt{Eq}\text{: } \chi \rhd^{\texttt{Eq}}_{\Gamma_i} \chi' \\ \Gamma_{i+1} = \Gamma_i[a_\$ \mapsto ([\texttt{f}^{\texttt{w}} : \chi', \overline{\texttt{f}^{\texttt{h}} : \chi'}], \texttt{C})][\chi \Rightarrow_{\Gamma_i} \chi'] \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1}\end{array}}{\text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i}^t}$$

Figure 4.9: Type rules for array related instructions

2. $\chi' = \texttt{Eq}(\chi)$,
3. $\chi' = null$,
4. $\Gamma_i(\chi') = [\cdots \texttt{f}_j^{\texttt{h}'_j} : \chi'_j \cdots]$ and $\Gamma_i(\chi) = [\cdots \texttt{f}_j^{\texttt{h}_j} : \chi_j \cdots]$ and, for every j, $\texttt{h}'_j \leqslant \texttt{h}_j$ and $\chi'_j \rhd^{\texttt{Eq}}_{\Gamma_i} \chi_j$.

Whenever there is Eq such that $\chi' \rhd^{\texttt{Eq}}_{\Gamma_i} \chi$ we also define $[\chi' \Rightarrow_{\Gamma_i} \chi]$ as follows:

- it is $\varnothing$ whenever $\chi' = \texttt{int} = \chi$ or $\chi' = \top = \chi$ or $\Gamma_i(\chi') = \Gamma_i(\chi)$,
- it is $\Gamma_i(\chi)$ whenever $\Gamma_i(\chi') = null$,
- it is $[\texttt{f}_1^{\texttt{w}} : \chi_1, \cdots, \texttt{f}_n^{\texttt{w}} : \chi_n][(\chi'_j \Rightarrow_{\Gamma_i} \chi_j)^{j \in 1..n}]$ whenever $\Gamma_i(\chi') = [\texttt{f}_1^{\texttt{h}'_1} : \chi'_1, \cdots, \texttt{f}_n^{\texttt{h}'_n} : \chi'_n]$ and $\Gamma_i(\chi) = [\texttt{f}_1^{\texttt{h}_1} : \chi_1, \cdots, \texttt{f}_n^{\texttt{h}_n} : \chi_n]$.

The JVML rules for manipulating single-dimensional arrays storing objects are defined in Figure 4.9.

The rule for anewarray defines *at the same time* a fresh name for the array, called a, the corresponding name $a_\$$ for the field value and a *representative* that defines its fields. The rule for aastore checks whether the object

χ to be stored in a is actually in the relation $a_\$ \rhd^{\mathtt{Eq}}_{\Gamma_i} \chi$, for some $\mathtt{Eq}$. This means that $a_\$$ is a representative of χ and this last object can be safely dismissed. The resulting environment Γ_{i+1} records that the array has been accessed in writing ($\mathtt{w}$) and *updates the names in* $\Gamma_i(\chi)$ *with the corresponding values in* $\Gamma_i(a_\$)$. The instruction `aastore` has a catastrophic side-effect on the argument χ. In fact, since names in $\Gamma_i(a_\$)$, become aliases of the corresponding names in $\Gamma_i(\chi)$, these aliases may modify the latter ones in an unpredictable way. Therefore, we need to *update* all the tree structures of $\Gamma_i(\chi)$ – *c.f.* the operation $[\chi \Mapsto_{\Gamma_i} a_\$]$. A similar situation occurs when one wants to modify a field of name $a_\$$. For this reason we also report in Figure 4.9 the rule of the `putfield` instruction when the name of the object to modify is indexed by a \$. The rule is different from the `putfield` in Figure 4.4.

There is another subtlety to examine before discussing the management of \$-indexed names in our algorithm for circularities in lams. This subtlety is about the merge operation $\oplus$ in Section 4.6 is made more restrictive when record types are arrays. Let

$$\mathtt{f}^{\mathtt{h}} : \chi \oplus_{\mathtt{aarr}} \mathtt{f}^{\mathtt{h}'} : \chi' \stackrel{def}{=} \begin{cases} \mathtt{f}^{\mathtt{h} \sqcup \mathtt{h}'} : \chi & \text{if } \chi = \chi' \\ \mathtt{f}^{\mathtt{w}} : \top & \text{otherwise} \end{cases}$$

Then, the operation $\mathtt{f}^{\mathtt{h}} : \chi \oplus \mathtt{f}^{\mathtt{h}'} : \chi'$ is refined into $\mathtt{f}^{\mathtt{h}} : \chi \oplus_{\mathtt{aarr}} \mathtt{f}^{\mathtt{h}'} : \chi'$ whenever either χ or χ' are \$-indexed names. In particular:

$$([\mathtt{idx}^{\mathtt{h}} : \chi], [\mathtt{C}]) \oplus ([\mathtt{idx}^{\mathtt{h}'} : \chi'], [\mathtt{C}]) \stackrel{def}{=} ([\mathtt{idx}^{\mathtt{h}} : \chi \oplus_{\mathtt{aarr}} \mathtt{idx}^{\mathtt{h}'} : \chi'], [\mathtt{C}])$$

Therefore, the access in parallel of two threads to a same array of objects is admitted only when the *two threads access to the same representative*. It is worth to notice that, representatives act as place holders – they may be instantiated by whatever name.

In fact, these \$-indexed names are managed in ad-hoc ways by the analyzer. In particular, a term like $(b, c)_t \& (c, a_\$)_{t'}$ is a circularity because $a_\$$ may be replaced by *every name of the same type*, including b, if its type is the same. Similarly, $(a_\$, b)_t \& (b, c)_t \& (a_\$, c)_{t'} \& (c, b)_{t'}$ is a circularity because the dependencies starting with $a_\$$ might concern different objects.

4.8.4 Recursive data types

Recursive data types are a standard feature of programming languages that allows programmers to define and use lists, trees, etc. These types are finite but not (statically) bounded and, as we did with arrays, we use finite representations.

A representative of a recursive type value is built by unfolding the recursive types (exactly) up to those nodes containing a name of a class already present in the tree. Nodes inside the tree are labeled by new names, nodes in the leaves are labeled either (for non recursive types) with $\top$ or `int` or with names already present in the environment or (for recursive types) with names subscribed by a \$, such as $r_{\$}$, where r is the label of the node of the class that is already present in the tree. By construction, these structures are finite. Names $r_{\$}$ have the same structure of r except that every name of a recursive type is indexed by \$ (and the access tag is set to `w`). For instance, if `C` is a class whose type is $[\texttt{val} : \texttt{int}, \texttt{next} : \texttt{C}]$ then, in correspondence of a `new C` instruction, we produce an environment

$$r \mapsto [\texttt{val}^{\texttt{w}} : \texttt{int}, \texttt{next}^{\texttt{w}} : r_{\$}], \quad r_{\$} \mapsto [\texttt{val}^{\texttt{w}} : \texttt{int}, \texttt{next}^{\texttt{w}} : r_{\$}]$$

(the access tags are all `w` because we are initializing the values).

We use the predicate "`C` *is recursive*" when `C` has a field whose values contain other values of class `C` or a subclass of its. Rules of `new` and `putfield` in Figure 4.4 are rewritten as follows when the class is recursive:

$$\frac{\begin{array}{c} P[i] = \texttt{new C} \quad \texttt{C} \text{ is recursive} \quad i+1 \in dom(P) \\ (a, \overline{a_{\$}}) = names(i) \quad \Gamma_{i+1} = \Gamma_i \oplus \texttt{Env}(\chi, \texttt{C}, \overline{a_{\$}}) \\ F_i = F_{i+1} \quad S_{i+1} = \chi \cdot S_i \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t}}$$

$$\frac{\begin{array}{c} P[i] = \texttt{putfield C.f : T} \quad \texttt{C} \text{ is recursive} \quad i+1 \in dom(P) \\ S_i = \chi' \cdot \chi \cdot S_{i+1} \quad \Gamma_i(\chi) = ([\texttt{f}^{\texttt{h}} : \chi_{\texttt{f}}, \overline{\texttt{f}^{\texttt{h}} : \chi_{\texttt{f}}}], \texttt{C}) \quad \text{there is } \texttt{Eq}\text{: } \chi' \rhd^{\texttt{Eq}}_{\Gamma_i} \chi_{\texttt{f}} \\ \Gamma_{i+1} = \Gamma_i[\chi \mapsto ([\texttt{f}^{\texttt{w}} : \chi_{\texttt{f}}, \overline{\texttt{f}^{\texttt{h}} : \chi_{\texttt{f}}}], \texttt{C})][\chi' \Mapsto_{\Gamma_i} \chi_{\texttt{f}}] \\ F_i = F_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \ \& \ \breve{\mathcal{R}}_i \ \& \ \widehat{Z_i^t}}$$

We omit any comment because the arguments are similar to those for arrays.

The presence of recursive types makes the definitions of the functions *flat*$(\cdot)$ and *mk_tree*$(\cdot)$ in Section 4.4 inadequate. Indeed, in the full language, the invocations to *flat*$(\cdot)$ and *mk_tree*$(\cdot)$ in the rules of Figures 4.4, 4.5, 4.7, and 4.8 are replaced by *flat*$_{\varnothing}(\cdot)$ and *mk_tree*$_{\varnothing}(\cdot)$, which are defined below:

$$
\mathit{flat}_A(\vartheta) = \begin{cases}
\varnothing & \text{if } \vartheta \in \{\top, \mathtt{int}, X\} \\
\varnothing & \text{if } \mathit{root}(\vartheta) = a_\$ \text{ and } a_\$ \in A \\
\mathit{flat}_{(A \setminus a) \cup \{a_\$\}}(\vartheta) & \text{if } \mathit{root}(\vartheta) = a_\$ \text{ and } a \in A \\
a \mapsto ([\overline{\mathtt{f}^\mathtt{h} : \mathit{root}(\vartheta)}], \mathtt{C}) \oplus (\bigoplus_{\vartheta' \in \overline{\vartheta}} \mathit{flat}_A(\vartheta')) & \text{if } \vartheta = (a[\overline{\mathtt{f}^\mathtt{h} : \vartheta}], \mathtt{C}) \\
 & \text{and } \mathtt{C} \textit{ is not recursive} \\
a \mapsto ([\overline{\mathtt{f}^\mathtt{h} : \mathit{root}(\vartheta)}], \mathtt{C}) \oplus (\bigoplus_{\vartheta' \in \overline{\vartheta}} \mathit{flat}_{A \cup \{a\}}(\vartheta')) & \text{if } \vartheta = (a[\overline{\mathtt{f}^\mathtt{h} : \vartheta}], \mathtt{C}) \\
 & \text{and } \mathtt{C} \textit{ is recursive and } a \notin A
\end{cases}
$$

$$
\mathit{mk\_tree}_A(\Gamma, \chi) = \begin{cases}
\chi & \text{if } \chi \in \{\top, \mathtt{int}, X\} \text{ or } (\chi = a_\$ \text{ and } a_\$ \in A) \\
(a[\overline{\mathtt{f} : \mathit{mk\_tree}_A(\Gamma, \chi)}], \mathtt{C}) & \text{if } \chi = a \text{ and } \Gamma(a) = ([\overline{\mathtt{f}^\mathtt{h} : \chi}], \mathtt{C}) \\
 & \text{and } \mathtt{C} \textit{ is not recursive} \\
(a[\overline{\mathtt{f} : \mathit{mk\_tree}_{A \cup \{a\}}(\Gamma, \chi)}], \mathtt{C}) & \text{if } \chi = a \text{ and } \Gamma(a) = ([\overline{\mathtt{f}^\mathtt{h} : \chi}], \mathtt{C}) \\
 & \text{and } a \notin A \text{ and } \mathtt{C} \textit{ is recursive} \\
(a_\$[\overline{\mathtt{f} : \mathit{mk\_tree}_{(A \setminus a) \cup \{a_\$\}}(\Gamma, \chi)}], \mathtt{C}) & \text{if } \chi = a_\$ \text{ and } \Gamma(a) = ([\overline{\mathtt{f}^\mathtt{h} : \chi}], \mathtt{C}) \\
 & \text{and } a \in A \text{ and } \mathtt{C} \textit{ is recursive}
\end{cases}
$$

4.8.5 Inheritance

$\mathtt{JVML}_d$ does not admit to derive classes from other classes. As a consequence, when a method is invoked, it is possible to uniquely locate the method definition. This feature is expressed in the type rule for `invokevirtual` in Figure 4.4 by the constraint that *typeof* (Γ_i, χ_0) *is always a single element.*

On the contrary, `Java`, as many object-oriented programming languages, admits subclasses, thereby allowing redefinitions of methods in the subclasses. In this context, locating a method definition is performed at run-time through a mechanism called dynamic dispatch that uses the actual value of the called object. Clearly, this mechanism has a significant impact on deadlock analysis because dispatching an invocation to a definition rather than another may change the overall result. It is also clear that, since our approach is static, our analyzer cannot resolve the dynamic dispatch in a precise way.

The type system discussed in this solution is over-constraining on the relations between consecutive environments Γ_i and Γ_{i+1} (or Γ_L, in case of jumps) because it imposes an equality relation between them. For instance, let `D` and `E` be subclasses of `C`, which has a method `foo`, which is overridden both in `D` and `E`. and consider the following code

```
C w;

if (z)
    w = new D x;
else
    w = new E y;

w.foo();
```

where the invocation `w.foo()` will be dispatched to the method `D.foo` or `E.foo`, according to the value of `z`. This code does not type with the system in Figures 4.4 and 4.5 because the classes of `x` and `y` are different (assuming that the function $names(\cdot)$ at instructions corresponding to `w = x;` and to `w = y;` returns the same name). There is a standard solution to this problem. This solution relies on subtypes and relaxes the equality constraint of types of successor instructions in Figures 4.4 and 4.5. This solution is described in more detail in Section 5.2 as it is closely related to the inference process. Here we focus on the case of inheritance, examining the solution for programs like the foregoing one.

Let $\mathsf{fields}(\mathtt{C})$ be the refinement of the function in Section 4.5 that returns a pair of tuples of field names $\bar{\mathtt{f}}, \bar{\mathtt{g}}$, where $\bar{\mathtt{f}}$ are the field names of the class `C` and $\bar{\mathtt{g}}$ are the names *defined* in one of $\mathtt{C}_1, \cdots, \mathtt{C}_h$ (and not inherited). Let $\bot$ be a new type value that means "*undefined*" (while $\top$ is the greatest value, $\bot$ is the least one). In our prototype, whenever an object is created, the corresponding name is bound to a record that contains the fields of the class *plus every field defined in its subclasses* (and not inherited). In particular, the rule that we use for `new C` is

$$\frac{\begin{array}{c} P[i] = \mathtt{new\ C} \qquad i+1 \in dom(P) \quad a = names(i) \quad \mathsf{fields}(\mathtt{C}) = \bar{\mathtt{f}}, \bar{\mathtt{g}} \\ \Gamma_i[a \mapsto ([\overline{\mathtt{f} : \top}, \overline{\mathtt{g} : \bot}], \mathtt{C})] <: \Gamma_{i+1} \\ F_i <: F_{i+1} \quad a \cdot S_i <: S_{i+1} \quad Z_i = Z_{i+1} \quad K_i = K_{i+1} \quad \mathcal{T}_i = \mathcal{T}_{i+1} \quad \mathcal{R}_i = \mathcal{R}_{i+1} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i^{\Gamma_i}} \,\&\, \widetilde{\mathcal{R}_i} \,\&\, \widehat{Z_i^t}}$$

where fields of class `C` are set to $\top$ while the other ones are set to $\bot$. We notice that the constraint between environments, fields and stacks is " $<:$ " rather than "=". This $<:$ relation makes it possible the analysis of programs like the above one. For example, of `C` has exactly one field `f`, `D` extends `C` with the field `g` and `E` extends `C` with the field $\mathtt{g}'$, `w` at `w.foo()` is typed with $([\mathtt{f} : \top, \mathtt{g} : \top, \mathtt{g}' : \top], \mathtt{C} \cdot \mathtt{D} \cdot \mathtt{E})$, where $\mathtt{C} \cdot \mathtt{D} \cdot \mathtt{E}$ denotes a set of classes.

The rule for `invokevirtual` that deals with dynamic dispatch on a set

of classes $\mathtt{C}_1 \cdots \mathtt{C}_h$ is (we are using again "=" rather than " <: "):

$$\frac{\begin{array}{c} P[i] = \texttt{invokevirtual}\ \ \mathtt{C}_1.\mathtt{m}\ (\mathtt{T}_0, \cdots, \mathtt{T}_k) \qquad i+1 \in \mathit{dom}(P) \qquad S_i = \chi_k \cdots\ \chi_0 \cdot S' \\ \mathit{typeof}(\Gamma_i, \chi_0) = \{\mathtt{C}_1, \cdots, \mathtt{C}_h\} \quad \mathit{mk\_tree}(\Gamma_i, (\chi_0, \cdots, \chi_k)) = (\tau_0, \cdots, \tau_k) \\ \Big(\textsc{bct}(\mathtt{C}_j.\mathtt{m})(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) = \langle \vartheta^j, \mathcal{T}^j, \mathcal{R}^j, K^j, (\vartheta^j_0, \cdots, \vartheta^j_k), \ell^j \rangle\Big)^{j \in 1..h} \\ \mathcal{T}' = \bigcup_{j \in 1..h} \mathcal{T}^j \quad \mathcal{R}' = \bigcup_{j \in 1..h} \mathcal{R}^j \quad K' = \bigcap_{j \in 1..h} K^j \\ \vartheta' = \vartheta^1 \oplus \cdots \oplus \vartheta^h \quad (\vartheta'_0, \cdots, \vartheta'_k) = (\vartheta^1_0, \cdots, \vartheta^1_k) \oplus \cdots \oplus (\vartheta^h_0, \cdots, \vartheta^h_k) \\ \Gamma_{i+1} = \Gamma_i[\mathit{flat}(\vartheta') \oplus (\bigoplus_{i \in 0..k} \mathit{flat}(\vartheta'_i))] \quad S_{i+1} = \mathit{root}(\vartheta') \cdot S' \\ F_{i+1} = F_i \qquad Z_{i+1} = Z_i \qquad K_{i+1} = K_i \cup K' \\ \mathcal{T}_{i+1}, \mathcal{R}_{i+1} = \begin{cases} (\mathcal{T}_i \backslash K') \cup \mathcal{T}',\ \mathcal{R}_i \cup \mathcal{R}' & \text{if } i \text{ is not recursive} \\ & \text{and } (\mathcal{T}_i \cup \mathcal{R}_i) \cap (\mathcal{T}' \cup \mathcal{R}') = \varnothing \\ \mathcal{T}_i \backslash K',\ (\mathcal{R}_i \backslash K') \cup \mathcal{T}' \cup \mathcal{R}' & \text{otherwise} \end{cases} \end{array}}{\textsc{bct}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P : \widehat{\mathcal{T}_i}\ \&\ \widecheck{\mathcal{R}_i}\ \&\ \widehat{Z_i^t}\ \&\ \textstyle\sum_{j \in 1..h} \mathtt{C}_j.\mathtt{m}(\tau_0, \cdots, \tau_k, t, \lceil Z_i \rceil) \to (\vartheta^j)^\triangledown}$$

In particular, for every class in the type of the carrier, we compute the corresponding lam and the overall type is the disjunction of the possible invocations – this is our modeling of *overridden methods* (we are assuming that a method that is not present in a superclass has empty behavior). It is worth to observe that the *cumulative effect* of all the overridden methods will impact on the judgment for the next instruction – see the definitions of Γ_{i+1}, F_{i+1}, S_{i+1}, K_{i+1}, $\mathcal{T}_{i+1}$, and $\mathcal{R}_{i+1}$.

The above rule assumes that the method `m` *is not overloaded* in any class. In case of overloading, we also need to consider all the possible instances caused by the types of the arguments.

Interfaces. In `Java` it is possible to define abstract classes, called *interfaces*, whose methods have empty behaviors. The language constrains the programmer to implement these interfaces by means of classes. We deal with interfaces as they were standard classes and with their implementations as they were subclasses of them. Henceforth `invokeinterface` instructions are typed as the `invokevirtual` instruction in the above rule.

4.8.6 Alternative concurrent models and native methods

During its evolution Java has been bringing different mechanisms to write concurrent programs. The current proposal targets the (classic way of) concurrency expressed by `Thread` objects and `synchronized` blocks (the `wait-notify` mechanism is not covered by this work and by our current prototype – see Section 5.4 for a discussion of a possible solution). There are as well many third party components that allow abstracting threads manipulation

in general. Our approach allows the user to manually instruct the analysis to deal with these scenarios. This is done by the addition of manual entries in the BCT, in this way method declarations with specific behaviors are not sent through the inference algorithm.

This approach makes also possible to deal with method implementations that do not comply with the restrictions described in this solution, such as native methods.

4.9 Full proof of correctness

In this section we develop the formal proof of the correctness of the type system presented in Section 4.5 and of its corresponding analysis presented in Section 4.6. The first epigraph of this section presents a short less formal overview of correctness demonstration. The full technical details of the demonstration can be found in the rest of the section.

4.9.1 Correctness overview

The first part of the proof addresses the soundness of the type system in Section 4.6. This requires

1. an operational semantics of JVML; we consider the one defined in Section 4.9.2;

2. an extension of the type system to handle JVM runtime configurations, defined in Section 4.9.6.

Then, as usual with type systems, the soundness is represented by a subject reduction theorem expressing that, if a JVM configuration cn has lam ℓ and cn reduces to a configuration cn' then (i) cn' is also well-typed and (ii) if ℓ' is the type of cn' then ℓ and ℓ' are in a relation called *later stage* (Section 4.9.5) and noted $\ell \succcurlyeq \ell'$.

The second part of the proof is about the correctness of the later stage relation with respect to the deadlock analysis. We first demonstrate that the lam of a deadlocked configuration always contains a *circular dependency* between names. Then we demonstrate that, if two types ℓ and ℓ' are in the later stage relationship, namely $\ell \succcurlyeq_{\text{BCT}} \ell'$, with respect to a given BCT, and ℓ *has no circular dependency* then also ℓ' has no circular dependency. The details of this proof can be found in Section 4.9.7.

As a byproduct of the above results we get that, if the lam of a JVML program has no circularity then the JVML program is deadlock-free. It is worth

to notice that our lams have infinite-state models. Nevertheless, in [34] has been demonstrated the decidability of whether the type will manifest a circular dependency in one of its states or not, here we extend that demonstration in Section 4.9.8.

4.9.2 The JVML formal semantics

In this section we present the operational semantics of $\texttt{JVML}_d$. Each instruction performs a transformation of machine states, that are configurations

$$P \Vdash_H \langle (pc^{\texttt{C.m}}, f_1, s_1) \cdot \varphi_1, z_1 \rangle_{t_1} \cdots \langle (pc^{\texttt{D.m}}, f_n, s_n) \cdot \varphi_n, z_n \rangle_{t_n} \dagger W$$

where each tuple $\langle \varphi, z \rangle_t$ represents an active thread t, such that:

- φ is the stack of activation records (or *frames*), where
  - *pc* is the *program counter* which contains the address of the instruction to be executed
  - f is a total map from the set of local variables to the set of values
  - s is the *operand Stack* of values
- z is the set of objects locked by the thread,

and W is the set of waiting threads (due to joins).

The *Heap* contains

- all objects created by `new`. Each object contains:
  - fields: the value of field a of object o is accessed with $H(o).a$. The update is noted $H(o).a \mapsto v$.
  - the class of the object, returned by $H(o).class$
  - a *wait set* containing the threads which are waiting for a thread to finish (namely they invoked `join` on o).
  - a counter $\digamma$ which tracks the number of locks acquired by a thread on the object

Moreover we use $H(o) \mapsto (\rho^{\texttt{C}}_{\perp}, \texttt{C})$ to allocate heap space for the object o of class `C`, by assigning a default value to the fields.

The *waiting set* $W = \{W_{o_1}, \ldots, W_{o_n}\}$, where W_{o_i} is the set of threads that performed a `join` on the object (of class `Thread`) o_i.

The rules are presented in Figures 4.10 and 4.11.

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{pop}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, v \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t} \text{(S-POP)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{push}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, 0 \cdot s) \cdot \varphi, z \rangle_t} \text{(S-PUSH)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{inc}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, n \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, (n+1) \cdot s) \cdot \varphi, z \rangle_t} \text{(S-INC)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{load}\ x}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, f(x) \cdot s) \cdot \varphi, z \rangle_t} \text{(S-LOAD)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{store}\ x}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, v \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f[x \mapsto v], s) \cdot \varphi, z \rangle_t} \text{(S-STORE)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{goto}\ L^{\mathtt{C.m}}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (L^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t} \text{(S-GOTO)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{putfield}\ \mathtt{D}.a\ \tau \quad H' = H(o).a \mapsto e}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, e \cdot o \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t} \text{(S-PUTFIELD)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{getfield}\ \mathtt{D}.a\ \tau \quad H(o).a = e}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, e \cdot s) \cdot \varphi, z \rangle_t} \text{(S-GETFIELD)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{new}\ \mathtt{D} \quad o \notin dom(H) \quad H' = H[o \mapsto (\rho^D, D)]}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z \rangle_t} \text{(S-NEW)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{if}\ L^{\mathtt{C.m}} \quad n \neq 0}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, n \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (L^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t} \text{(S-IF-TRUE)}$$

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{if}\ L^{\mathtt{C.m}}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, 0 \cdot s) \cdot \varphi, z) \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t} \text{(S-IF-FALSE)}$$

Figure 4.10: Reduction rules: Basic operators

4.9.3 Lam semantics

Let $\mathcal{A}$ be the (infinite) set of *object names*, ranged over by a, b, c, $\cdots$. In the syntax of ℓ, see Section 4.4, the operations "&" and "+" are associative, commutative with $\mathsf{0}$ being the identity on &, and definitions and lams are equal up-to alpha renaming of bound names. Namely, if $a \notin var(\ell)$, the

(S-INVK)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual\ D.m'}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \quad \mathtt{synchronized} \notin mod(\mathtt{D.m})}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (1^{\mathtt{D.m'}}, f_2[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon) \cdot (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \rangle_t}$$

(S-INVK-SYNCH-0)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual\ D.m'}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \quad \mathtt{synchronized} \in mod(\mathtt{D.m'}) \quad H(o).\mathcal{F} = 0 \quad H^i = H(o).\mathcal{F} \mapsto 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \backslash \{o\} \rangle_t \rightarrow \Vdash_H \langle (1^{\mathtt{D.m'}}, f_2[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon) \cdot (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \rangle_t}$$

(S-INVK-SYNCH-N)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{invokevirtual\ D.m'}(\mathtt{T}_1, \ldots, \mathtt{T}_n) \quad \mathtt{synchronized} \in mod(\mathtt{D.m'}) \quad H(o).\mathcal{F} = n,\ n > 0 \quad H' = H(o).\mathcal{F} \mapsto n + 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \cup \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (1^{\mathtt{D.m'}}, f_2[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon) \cdot (pc^{\mathtt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \cup \{o\} \rangle_t}$$

(S-START)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{start\ D}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t\ , \langle (1^{java/lang/Thread.run}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o}$$

(S-JOIN)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{join} \quad W_o' = W_o \uplus \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi_{t_1}, z_{t_1} \rangle_t^o}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi_{t_1}, z_{t_1} \rangle_t\ , \langle \varphi_{t2} \cdot (pc^{java/lang/Thread.run}, f_1^{t2}, s_1^{t2}), z_{t_2} \rangle_o \dagger W_o \rightarrow \Vdash_H \langle \varphi_{t2} \cdot (pc^{java/lang/Thread.run}, f_1^{t2}, s_1^{t2}), z_{t_2} \rangle_o \dagger W_o'}$$

(S-RETURN-RUN)
$$\frac{P[pc^{java/lang/Thread.run}] = \mathtt{return}}{P \Vdash_H \langle (pc^{java/lang/Thread.run}, f, s), z_1 \rangle_o \dagger \varnothing_o^j \uplus (\langle (pc_k, f_k, s_k) \cdot \varphi_k, z_k) \rangle_t^o)^{k \in K} \rightarrow \Vdash_H (\langle (pc_k, f_k, s_k) \cdot \varphi_k, z_k) \rangle_o)^{k \in K} \dagger \varnothing_o^j}$$

(S-RETURN)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{return} \quad \mathtt{synchronized} \notin mod(\mathtt{C.m})}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle pc^{\mathtt{D.m}}, f_2, v \cdot s_2) \cdot \varphi, z \rangle_t}$$

(S-RETURN-SYNCH-0)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{return} \quad \mathtt{synchronized} \in mod(\mathtt{C.m}) \quad H(o).\mathcal{F} = 1 \quad H' = H(o).\mathcal{F} \mapsto 0}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \uplus \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{D.m}}, v \cdot f_2, s_2) \cdot \varphi, z \rangle_t}$$

(S-RETURN-SYNCH-N)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{return} \quad \mathtt{synchronized} \in mod(\mathtt{C.m}) \quad H(o).\mathcal{F} = n, n > 1 \quad H' = H(o).\mathcal{F} \mapsto n - 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\mathtt{D.m}}, \langle f_2, \bullet \cdot s_2) \cdot \varphi, z \cup \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{D.m}}, f_2, v \cdot s_2 \cdot \varphi, z \cup \{o\} \rangle_t}$$

(S-MONITOREXIT-0)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{monitorexit} \quad H(o).\mathcal{F} = 1 \quad H' = H(o).\mathcal{F} \mapsto 0}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \uplus \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t}$$

(S-MONITOREXIT-N)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{monitorexit} \quad H(o).\mathcal{F} = n,\ n > 1 \quad H' = H(o).\mathcal{F} \mapsto n - 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \cup \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \cup \{o\} \rangle_t}$$

(S-MONITORENTER-0)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{monitorenter} \quad H = H(o).\mathcal{F} = 0 \quad H' = H(o).\mathcal{F} \rightarrow\ 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \backslash \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \uplus \{o\} \rangle_t}$$

(S-MONITORENTER-N)
$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{monitorenter} \quad H = H(o).\mathcal{F} = n,\ n > 0 \quad H' = H(o).\mathcal{F} \rightarrow\ n + 1}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, o \cdot s) \cdot \varphi, z \cup \{o\} \rangle_t \rightarrow \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \cup \{o\} \rangle_t}$$

Figure 4.11: Reduction rules: Invocations and synchronizations

following axioms hold:

$$(\nu\, a)\ell = \ell \qquad ((\nu\, a)\ell')\&\ell = (\nu\, a)(\ell'\&\ell) \qquad ((\nu\, a)\ell') + \ell = (\nu\, a)(\ell' + \ell)$$

Additionally, when $\mathtt{V}$ ranges over lams that do not contain function invocations, the following axioms hold:

$$\mathtt{V}\&\mathtt{V} = \mathtt{V} \qquad \mathtt{V} + \mathtt{V} = \mathtt{V} \qquad \mathtt{V}\&(\ell' + \ell'') = \mathtt{V}\&\ell' + \mathtt{V}\&\ell'' \tag{4.1}$$

These axioms permit to rewrite a lam *without function invocations* as a *collection* (operation +) *of relations* (elements of a relation are gathered by the operation &). Let $\equiv$ be the least congruence containing the above axioms. (The axioms (4.1) are restricted to terms $\mathtt{V}$ that do not contain function invocations because $\mathtt{m}(\overline{\tau}) \to \tau'\&((a,b)_t + (b,c)_t) \neq (\mathtt{m}(\overline{\tau}) \to \tau'\&(a,b)_t) + (\mathtt{m}(\overline{\tau}) \to \tau'\&(b,c)_t)$ because the evaluation of the two lams (see below) may produce terms with different names.)

Operational semantics. Let a *lam context*, noted $\mathtt{L}[\,]$, be a term derived by the following syntax:

$$\mathtt{L}[\,] \quad ::= \quad [\,] \quad | \quad \ell\&\mathtt{L}[\,] \quad | \quad \ell + \mathtt{L}[\,]$$

As usual $\mathtt{L}[\ell]$ is the lam where the hole of $\mathtt{L}[\,]$ is replaced by ℓ. According to the syntax, lam contexts have no ν-binder; that is, the hassle of name captures is avoided. The operational semantics of a program $\big(\mathcal{L}, \ell\big)$ is a transition system where *states* are lams, the *transition relation* is the least one satisfying the rule

$$\frac{\mathtt{m}(\overline{\tau}) \to \tau' = (\nu\, \overline{c})\ell_{\mathtt{m}} \in \mathcal{L} \quad (\overline{\tau} \to \tau')\sigma = \overline{\tau}'' \to \tau''' \quad \overline{c}' \text{ fresh} \quad (\ell_{\mathtt{m}}\{\overline{c}'/\overline{c}\})\sigma = \ell'_{\mathtt{m}}}{\mathtt{L}[\mathtt{m}(\overline{\tau}'') \to \overline{\tau}'''] \to \mathtt{L}[\ell'_{\mathtt{m}}]}$$

and the initial state is the lam ℓ' such that $\ell \equiv (\nu\, \overline{c})\ell'$ and ℓ' does not contain any ν-binder, as well as the lam $\ell_{\mathtt{m}}$. (The class name in the names of lam functions has been dropped, for simplicity.) We write $\to^*$ for the reflexive and transitive closure of $\to$.

By (RED), a lam ℓ is evaluated by successively replacing function invocations with the corresponding lam instances. Name creation is handled by replacing bound names of function bodies with fresh names.

$$\text{(L-0)}\quad \ell \geqslant 0 \qquad \text{(L-RES)}\quad \frac{\ell \geqslant \ell'}{(\nu a)\ell \geqslant (\nu a)\ell'} \qquad \text{(L-PLUS)}\quad \frac{\ell_1 \geqslant \ell_1' \quad \ell_2 \geqslant \ell_2'}{\ell_1 + \ell_1' \geqslant \ell_2 + \ell_2'} \qquad \text{(L-AND)}\quad \frac{\ell_1 \geqslant \ell_1' \quad \ell_2 \geqslant \ell_2'}{\ell_1 \,\&\, \ell_1' \geqslant \ell_2 \,\&\, \ell_2'}$$

$$\text{(L-INVK)}\quad \frac{\text{BCT}(\texttt{C.m}) = (\bar{\tau}, t, a) \to (\nu\, \overline{a'})\langle \vartheta, \mathcal{T}, \mathcal{R}, K, \overline{\vartheta}, \ell\rangle \quad \overline{b}\ \textit{fresh}}{\texttt{C.m}(a[\texttt{f} : \tau], \overline{\tau'}) \to \vartheta^{\triangledown} \geqslant \ell\{\overline{b}/\overline{a'}\}}$$

Figure 4.12: The later-stage relation

4.9.4 Flattening and circularities

Informally, a lam has a *circularity* when it has a conjunction of dependencies whose transitive closure contains a pair $(a, a)_{\checkmark}$ (see Section 4.6). Here the hassle is that lams cannot be always rewritten in a form suitable for defining circularities, namely disjunctions of conjunctions, because the equations (4.1) do not apply to terms containing invocations. To overcome this problem, we define the operation of flattening. The transitive closure of a lam has been defined in Section 4.6.

Definition 9.1. *The* flattening *of a lam ℓ, noted $(\ell)^\flat$, is the lam ℓ where every function invocation has been replaced by* $\mathsf{0}$.

For example, let $\ell = \mathtt{m}(a, b, c, t) + (a, b)_t \& \mathtt{m}'(b, c, t') \& \mathtt{m}(d, b, c, t)$ (we assume that return types of lam functions are empty). Then $(\ell)^\flat = \mathsf{0} + (a, b)_t \& \mathsf{0} \& \mathsf{0}$.

Definition 9.2. *A lam ℓ has a* circularity *if $(\ell)^\flat \equiv \sum_{i\in I} \ell_i$, where every ℓ_i is a conjunction of dependencies, and, for some a and i, $(a, a)_{\checkmark}$ is a subterm of ${\ell_i}^+$.*

A lam program $\big(\mathcal{L}, \ell\big)$ has a circularity *if there exists ℓ' such that $\ell \longrightarrow {}^*\ell'$ and ℓ' has a circularity.*

4.9.5 Later stage relation

In the following subject reduction theorem we use a relation, called *later-stage relation* $\geqslant$, which is the least congruence with respect to lams that contains the rules presented in Figure 4.12. The notation $\geqslant$ assumes the presence of a behavioural class table BCT – we should have noted the later-stage relation as $\geqslant_{\text{BCT}}$, but we prefer to keep the BCT implicit. A relevant property of the later stage relation is the following.

Lemma 4.9.1 (Circularities)**.** *Let $\ell \geqslant \ell'$. If ℓ' has a circularity then ℓ has also a circularity.*

This result mostly follows from the formal results of [34, 48].

4.9.6 Runtime typing

Definition 9.3 (Γ-H Agreement)**.** *An environment* Γ agrees with *a heap* H*, written* $\Gamma \approx H$ *if*

- $o \in dom(H)$ *implies* $o \in dom(\Gamma)$ *and*
- $H(o).class = \mathtt{C}$ *and* $H(o).\mathtt{f}_i \mapsto v_i$*, for* $i \in 1..n$*, imply* $\Gamma = o \mapsto (\big[\mathtt{f}_1^{\mathtt{h}_1} : \chi_1, \cdots, \mathtt{f}_i^{\mathtt{h}_i} : \chi_i, \cdots, \mathtt{f}_n^{\mathtt{h}_n} : \chi_n\big], \mathtt{C})$*, where, if* v_i *is an object name, then* $v_i = \chi_i$*.*

Definition 9.4 (S/F/Z-Agreements)**.** *Given* Γ*, we define the following agreements between the static environments* S*,* F *and* Z *and the corresponding dynamic ones* s*,* f*, and* z*.*

S-agreement.

- $\epsilon \approx_\Gamma \epsilon$,
- $\chi \cdot S \approx_\Gamma \bullet \cdot s$ *, if* $S \approx_\Gamma s$,
- $\chi \cdot S \approx_\Gamma v \cdot s$ *, if* $S \approx_\Gamma s$ *and* $\Gamma \vdash v : \chi$*.*

F-agreement.

- $\varnothing \approx_\Gamma \varnothing$,
- $F\big[x \mapsto \chi\big] \approx_\Gamma f\big[x \mapsto v\big]$ *, if* $F \approx_\Gamma f$ *and* $\Gamma \vdash v : \chi$*.*

Z-agreement.

- $\epsilon \approx_\Gamma \epsilon$,
- $o \cdot Z \approx o \cdot z$ *if* $Z \approx z$

Definition 9.5 (Reachable instructions set)**.** *Let* $reachable(P, i)$*, called the set of addresses reachable from the instruction* $i \in dom(P)$*, be the least set satisfying the following equations:*

$$
\begin{array}{lcll}
reachable(P,i) & = & \varnothing & \text{if } P\big[i\big] = \texttt{return} \\
reachable(P,i) & = & \{L\} \cup reachable(P,L) & \text{if } P\big[i\big] = \texttt{goto L} \\
reachable(P,i) & = & \{i+1, L\} \cup reachable(P,i+1) \cup reachable(P,L) & \text{if } P\big[i\big] = \texttt{if L} \\
reachable(P,i) & = & \{i+1\} \cup reachable(P,i+1) & \text{otherwise}
\end{array}
$$

The typing judgement for configurations is

$$P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \mathfrak{C}) : \ell$$

that is defined by the following two rules:

(T-CONF-AND)

$$\frac{\begin{array}{c}\left(P, \mathrm{BCT}, \Gamma^{(h)}, F^{(h)}, S^{(h)}, Z^{(h)}, \mathcal{T}^{(h)}, \mathcal{R}^{(h)}, K^{(h)} \vdash \langle \varphi_h, z_h \rangle_{t_h} : \ell_h\right)^{h \in 1..n} \\ \Gamma = \bigcup_{h \in 1..n} \Gamma^{(h)} \quad F = F^{(1)} \cdots F^{(n)} \quad S = S^{(1)} \cdots S^{(n)} \quad Z = Z^{(1)} \cdots Z^{(n)} \\ \mathcal{T} = \bigcup_{h \in 1..n} \mathcal{T}^{(h)} \quad \mathcal{R} = \bigcup_{h \in 1..n} \mathcal{R}^{(h)} \quad K = \bigcup_{h \in 1..n} K^{(h)} \\ \Gamma \approx H\end{array}}{P, \mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C}_1.\mathtt{m}_1}, f_1, s_1) \cdot \varphi_1, z_1 \rangle_{t_1} \cdots \langle (pc^{\mathtt{C}_n.\mathtt{m}_n}, f_n, s_n) \cdot \varphi_n, z_n \rangle_{t_n}) : \&_{i \in 1..n} \ell_i}$$

(T-CONF-SINGLE)

$$\frac{\begin{array}{c}\left(\mathrm{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i;\ \Psi_i\right)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \\ P, \mathrm{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'' \vdash \langle \varphi, (z \backslash Z') \rangle_t : \ell \quad \Gamma'' = \Gamma' \cup \Gamma'' \\ F = F' \cdot F'' \quad S = S' \cdot S'' \quad Z = Z' \cdot Z'' \quad \mathcal{T} = \mathcal{T}' \cup \mathcal{T}'' \quad \mathcal{R} = \mathcal{R}' \cup \mathcal{R}'' \quad K = K' \cup K'' \\ Z \approx z \quad S'_{pc^{\mathtt{C.m}}} \approx s \quad F'_{pc^{\mathtt{C.m}}} \approx f\end{array}}{P, \mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t : \sum_{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \ell_i + \ell}$$

4.9.7 Subject Reduction

Lemma 4.9.2. *Let* C.m *be a method of a* $\mathtt{JVML}_d$ *program. If*

$$\mathrm{BCT}(\mathtt{C.m})(\overline{\tau}, t, a)(\overline{a'}) = \langle \vartheta, \mathcal{T}', \mathcal{R}', K', \overline{\vartheta}, \ell \rangle \,.$$

then

$$\left(\mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K, i \vdash_t P[\mathtt{C.m}] : \ell_i\,;\ \Psi_i\right)^{i \in reachable(P[\mathtt{C.m}], 1^{\mathtt{C.m}})},$$

where

$F_1 = F_\top[0 \mapsto root(\tau_0), \cdots, k \mapsto root(\tau_k)]$,
$\Gamma_1 = \bigoplus_{i \in 0..k} flat(\tau_i)$,
$S_1 = \varepsilon$,
$Z_1 = A$, (*if* C.m *is synchronized* $A = root(\tau_0) \cdot a$, *otherwise* $A = a$)
$\mathcal{T}_1 = \varnothing$,
$\mathcal{R}_1 = \varnothing$,
$K_1 = \varnothing$,
$\ell = \sum_{i \in reachable(P[\mathtt{C.m}], 1^{\mathtt{C.m}})} \ell_i$,
$\bigsqcup_{i \in reachable(P[\mathtt{C.m}], 1^{\mathtt{C.m}})} \Psi_i = (\vartheta, A, \mathcal{T}', \mathcal{R}', K', \Gamma')$,
$\overline{\vartheta} = mk\_tree(\Gamma', (root(\tau_0), \cdots, root(\tau_k)))$, *and*
$\overline{a'} = fn(\vartheta, \mathcal{T}', \mathcal{R}', K', \overline{\vartheta}) \backslash\ fn(\tau_0, \cdots, \tau_k, t, a)$.

Theorem 4.9.3 (Subject Reduction)**.** *If* $P, \mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \mathfrak{C}) : \ell$ *and* $\Gamma \approx H$ *and* $\Vdash_H \mathfrak{C} \to \Vdash_{H'} \mathfrak{C}'$, *then there exists* ℓ' *and* Γ', F', S', Z', $\mathcal{T}'$,$\mathcal{R}'$,K' *such that* $\Gamma' \approx H'$ *and* $P, \mathrm{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K' \vdash (\Vdash_{H'} \mathfrak{C}') : \ell'$ *and* $\ell \geqslant \ell'$.

Proof. (*Sketch*) By case analysis on the last reduction rule used.

Case `pop`. We have $P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\texttt{C.m}}, f, v \cdot s) \cdot \varphi, z \rangle_t) : \ell$ and

$$\frac{P[pc^{\texttt{C.m}}] = \texttt{pop}}{P \Vdash_H \langle (pc^{\texttt{C.m}}, f, v \cdot s) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\texttt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t} \text{(S-POP)}$$

By the configuration typing rules, we get

$$\begin{gathered}(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})} \\ P, \text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'' \vdash \langle \varphi, z \rangle_t : \ell_\varphi \quad \Gamma = \Gamma' \cup \Gamma'' \quad F = F' \cdot F'' \\ S = S' \cdot S'' \quad Z = Z' \cdot Z'' \quad \mathcal{T} = \mathcal{T}' \cup \mathcal{T}'' \quad \mathcal{R} = \mathcal{R}' \cup \mathcal{R}'' \quad K = K' \cup K'' \\ Z'_{pc^{\texttt{C.m}}} \approx z \quad S'_{pc^{\texttt{C.m}}} \approx v \cdot s \quad F'_{pc^{\texttt{C.m}}} \approx f\end{gathered}$$

From $(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})}$ and Definition 9.5 we derive that

$$\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', pc^{\texttt{C.m}} \vdash_t P[\texttt{C.m}] : \ell_{pc^{\texttt{C.m}}}$$

and

$$(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)}.$$

Therefore $\ell = (\ell_{pc^{\texttt{C.m}}} + (\sum \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)}) + \ell_\varphi$. By the typing rule for `pop`: $S'_{pc^{\texttt{C.m}}} = \chi \cdot S'_{pc^{\texttt{C.m}}+1}$; therefore, since $S'_{pc^{\texttt{C.m}}} \approx v \cdot s$ then $S'_{pc^{\texttt{C.m}}+1} \approx s$, by Definition 9.4.

We can apply again the configuration typing rules and we get that $\ell' = (\sum \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)} + \ell_\varphi$, thus $\ell \geqslant \ell'$.

Case `invokevirtual`.

We have $P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\texttt{C.m}}, f, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s) \cdot \varphi, z \rangle_t) : \ell$ and then one of three operational rules may have been applied: (S-INVK), (S-INVK-SYNCH-0), or (S-INVK-SYNCH-N). Let us consider the former:

$$\frac{\begin{gathered}P[pc^{\texttt{C.m}}] = \texttt{invokevirtual D.m}'(\texttt{T}_1, \ldots, \texttt{T}_n) \\ \texttt{synchronized} \notin mod(\texttt{D.m}')\end{gathered}}{\begin{gathered}P \Vdash_H \langle (pc^{\texttt{C.m}}, f_1, v_n \cdot \cdots \cdot v_1 \cdot o \cdot s_1) \cdot \varphi, z \rangle_t \rightarrow \\ \Vdash_H \langle (1^{\texttt{D.m}'}, f_2[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n], \epsilon) \cdot (pc^{\texttt{C.m}} + 1, f_1, \bullet \cdot s_1) \cdot \varphi, z \rangle_t\end{gathered}} \text{(S-INVK)}$$

By the configuration typing rules, we get

$$\begin{array}{c}(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\![\texttt{C.m}]\!] : \ell_i)^{i \in reachable(P[\![\texttt{C.m}]\!], pc^{\texttt{C.m}})} \\ P, \text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'' \vdash \langle \varphi, z \rangle_t : \ell_\varphi \quad \Gamma = \Gamma' \cup \Gamma'' \\ F = F' \cdot F'' \quad S = S' \cdot S'' \quad Z = Z' \cdot Z'' \quad \mathcal{T} = \mathcal{T}' \cup \mathcal{T}'' \quad \mathcal{R} = \mathcal{R}' \cup \mathcal{R}'' \\ K = K' \cup K'' \quad Z'_{pc^{\texttt{C.m}}} \approx z \quad S'_{pc^{\texttt{C.m}}} \approx v_n \cdot \cdots \cdot v_1 \cdot o \cdot s \quad F'_{pc^{\texttt{C.m}}} \approx f\end{array}$$

From $(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\![\texttt{C.m}]\!] : \ell_i)^{i \in reachable(P[\![\texttt{C.m}]\!], pc^{\texttt{C.m}})}$ and Definition 9.5 we derive that

$$\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', pc^{\texttt{C.m}} \vdash_t P[\![\texttt{C.m}]\!] : \ell_{pc^{\texttt{C.m}}}$$

and

$$(\text{BCT}, \Gamma', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'', i \vdash_t P[\![\texttt{C.m}]\!] : \ell_i)^{i \in reachable(P[\![\texttt{C.m}]\!], pc^{\texttt{C.m}}+1)}.$$

Therefore

$$\ell = \ell_{pc^{\texttt{C.m}}} + (\sum \ell_i)^{i \in reachable(P[\![\texttt{C.m}]\!], pc^{\texttt{C.m}}+1)} + \ell_\varphi. \tag{4.2}$$

We assume that C.m is recursive (this case and that where $pc^{\texttt{C.m}}$ is inside an iteration are the interesting cases). By (T-INVK) typing rule:

$$\ell_{pc^{\texttt{C.m}}} = \widehat{\mathcal{T}'_{pc^{\texttt{C.m}}}} \ \& \ \widetilde{\mathcal{R}'_{pc^{\texttt{C.m}}}} \ \& \ \widehat{Z'^{\,t}_{pc^{\texttt{C.m}}}} \ \& \ \texttt{D.m}'(\tau_1, \cdots, \tau_n, t, \lceil Z'_{pc^{\texttt{C.m}}} \rceil) \to \vartheta,$$

where

$$\begin{array}{c}S'_{pc^{\texttt{C.m}}} = \chi_n \cdots \ \chi_1 \cdot S''' \quad typeof(\Gamma'_{pc^{\texttt{C.m}}}, \chi_1) = \texttt{D} \\ mk\_tree(\Gamma'_{pc^{\texttt{C.m}}}, (\chi_1, \cdots, \chi_n)) = (\tau_1, \cdots, \tau_n) \quad \overline{b} = names(pc^{\texttt{C.m}}) \\ \text{BCT}(\texttt{D.m}')(\tau_1, \cdots, \tau_n, t, \lceil Z'_{pc^{\texttt{C.m}}} \rceil)(\overline{b}) = \langle \vartheta, \mathcal{T}''', \mathcal{R}''', K''', (\vartheta_1, \cdots, \vartheta_n), \ell_{\texttt{D.m}'} \rangle \\ \Gamma'_{pc^{\texttt{C.m}}+1} = \Gamma'_{pc^{\texttt{C.m}}}[flat(\vartheta) \oplus (\bigoplus_{pc^{\texttt{C.m}} \in 1..n} flat(\vartheta_{pc^{\texttt{C.m}}}))] \\ S'_{pc^{\texttt{C.m}}+1} = root(\vartheta) \cdot S''' \quad F'_{pc^{\texttt{C.m}}+1} = F'_{pc^{\texttt{C.m}}} \quad Z'_{pc^{\texttt{C.m}}+1} = Z'_{pc^{\texttt{C.m}}} \quad K'_{pc^{\texttt{C.m}}+1} = K'_{pc^{\texttt{C.m}}} \cup K''' \\ \mathcal{T}'_{pc^{\texttt{C.m}}+1}, \mathcal{R}'_{pc^{\texttt{C.m}}+1} = (\mathcal{T}'_{pc^{\texttt{C.m}}} \backslash K''') \cup \mathcal{T}''', \ (\mathcal{R}'_{pc^{\texttt{C.m}}} \backslash K''') \cup \mathcal{R}'''\end{array}$$

By (T-INVK) typing rule $S'_{pc^{\texttt{C.m}}} = \chi_n \cdots \ \chi_1 \cdot S'''$ and $S'_{pc^{\texttt{C.m}}+1} = root(\vartheta) \cdot S'''$ so since $S'_{pc^{\texttt{C.m}}} \approx v_n \cdot \cdots \cdot v_1 \cdot o \cdot s$ then $S'_{pc^{\texttt{C.m}}+1} \approx \bullet \cdot s$.

Therefore we obtain that

$$P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash \langle (pc^{\texttt{C.m}} + 1, f, \bullet \cdot s) \cdot \varphi, z \rangle_t : \ell_{pc^{\texttt{C.m}}+1} \tag{4.3}$$

where $\ell_{pc^{\texttt{C.m}}+1} = (\sum \ell_i)^{i \in reachable(P[\![\texttt{C.m}]\!], pc^{\texttt{C.m}}+1)} + \ell_\varphi$

By Lemma 4.9.2,

$$\left(\text{BCT}, \Gamma^{\texttt{D.m}'}, F^{\texttt{D.m}'}, S^{\texttt{D.m}'}, Z^{\texttt{D.m}'}, \mathcal{T}^{\texttt{D.m}'}, \mathcal{R}^{\texttt{D.m}'}, K^{\texttt{D.m}'}, i \vdash_t P[\![\texttt{D.m}']\!] : \ell_i^{\texttt{D.m}'} \ ; \ \Psi_i\right)^{i \in reachable(P[\![\texttt{D.m}']\!], 1^{\texttt{D.m}'})},$$

where

$$
\begin{array}{l}
F_1^{\mathtt{D.m'}} = F_\top^{\mathtt{D.m'}}[0 \mapsto root(\tau_1), \cdots, n \mapsto root(\tau_n)],\\
\Gamma_1^{\mathtt{D.m'}} = \bigoplus_{i \in 1..n} flat(\tau_i),\\
S_1^{\mathtt{D.m'}} = \varepsilon,\\
Z_1^{\mathtt{D.m'}} = \lceil Z'_{pc^{\mathtt{C.m}}} \rceil,\\
\mathcal{T}_1^{\mathtt{D.m'}} = \varnothing,\\
\mathcal{R}_1^{\mathtt{D.m'}} = \varnothing,\\
K_1^{\mathtt{D.m'}} = \varnothing,\\
\ell^{\mathtt{D.m'}} = \sum_{i \in reachable(P[\mathtt{D.m'}], 1^{\mathtt{D.m'}})} \ell_i^{\mathtt{D.m'}},\\
\bigsqcup_{i \in reachable(P[\mathtt{D.m'}], 1^{\mathtt{D.m'}})} \Psi_i = (\vartheta, \lceil Z'_{pc^{\mathtt{C.m}}} \rceil, \mathcal{T}'^{\mathtt{D.m'}}, \mathcal{R}'^{\mathtt{D.m'}}, K'^{\mathtt{D.m'}}, \Gamma'^{\mathtt{D.m'}}),\\
\overline{\vartheta} = mk\_tree(\Gamma'^{\mathtt{D.m'}}, (root(\tau_1), \cdots, root(\tau_n))), and\\
\overline{b} = fn(\vartheta, \mathcal{T}'^{\mathtt{D.m'}}, \mathcal{R}'^{\mathtt{D.m'}}, K'^{\mathtt{D.m'}}, \overline{\vartheta}) \setminus fn(\tau_1, \cdots, \tau_n, t, a).
\end{array}
$$

Thus

$$
\begin{array}{l}
P, \text{BCT}, \Gamma^{\mathtt{D.m'}} \cup \Gamma, F^{\mathtt{D.m'}} \cdot F, S^{\mathtt{D.m'}} \cdot S, Z^{\mathtt{D.m'}} \cdot Z, \mathcal{T}^{\mathtt{D.m'}} \cup \mathcal{T}'^{\mathtt{D.m'}}, \mathcal{R}^{\mathtt{D.m'}} \cup \mathcal{R}'^{\mathtt{D.m'}}, K^{\mathtt{D.m'}} \cup K \vdash\\
\quad (\langle\!\langle (1^{\mathtt{D.m'}}, f'[0 \mapsto o, 1 \mapsto v_1, \ldots, n \mapsto v_n, this \mapsto o], \epsilon)\\
\qquad \cdot (pc^{\mathtt{C.m}} + 1, f, s) \cdot \varphi, z \rangle_t) : \ell^{\mathtt{D.m'}} + \ell_{pc^{\mathtt{C.m}}+1}
\end{array}
$$

and $\ell \geqslant \ell^{\mathtt{D.m'}} + \ell_{pc^{\mathtt{C.m}}+1}$

The last step is to prove that $\Gamma^{\mathtt{D.m'}} \cup \Gamma \approx H$, which follows directly from Definition 9.3 and the fact that $\Gamma \approx H$.

The other two cases of the operational semantics are proven in a similar way, except that we may have to check the agreement of the modified environment z, but this is straightforward from the definition of agreement and the typing rule for configuration.

Case `start`. We have $P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t) : \ell$ and then

(S-START)

$$
\frac{P[pc^{\mathtt{C.m}}] = \mathtt{start\ D}}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f_1, o \cdot s_1) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle (pc^{\mathtt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t \ , \langle (1^{\mathtt{D.run}}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o}
$$

By the configuration typing rules, we obtain

$$
\begin{array}{c}
(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}\\
P, \text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'' \vdash \langle \varphi, z \rangle_t : \ell_\varphi \quad \Gamma = \Gamma' \cup \Gamma''\\
F = F' \cdot F'' \quad S = S' \cdot S'' \quad Z = Z' \cdot Z'' \quad \mathcal{T} = \mathcal{T}' \cup \mathcal{T}'' \quad K = K' \cup K''\\
Z'_{pc^{\mathtt{C.m}}} \approx z \quad S'_{pc^{\mathtt{C.m}}} \approx o \cdot s \quad F'_{pc^{\mathtt{C.m}}} \approx f
\end{array}
$$

From $(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})}$ and Definition 9.5 we derive that

$$\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', pc^{\texttt{C.m}} \vdash_t P[\texttt{C.m}] : \ell_{pc^{\texttt{C.m}}}$$

and

$$(\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)}.$$

Therefore

$$\ell = \ell_{pc^{\texttt{C.m}}} + (\sum \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)} + \ell_\varphi. \tag{4.4}$$

By (T-START) typing rule:

$$\ell_{pc^{\texttt{C.m}}} = \widehat{\mathcal{T}'_{pc^{\texttt{C.m}}}} \ \&\ \widetilde{\mathcal{R}'_{pc^{\texttt{C.m}}}} \ \&\ \widehat{Z'^{\,t}_{pc^{\texttt{C.m}}}}$$

where

$$S'_{pc^{\texttt{C.m}}} = o \cdot S'_{pc^{\texttt{C.m}}+1}$$
$$typeof(\Gamma'_{pc^{\texttt{C.m}}}, o) = \texttt{D} \quad mk\_tree(\Gamma'_{pc^{\texttt{C.m}}}, o) = \tau \quad \bar{b} = names(pc^{\texttt{C.m}})$$
$$\text{BCT}(\texttt{D.run})(\tau, o, lock_o)(\bar{b}) = \langle void, \mathcal{T}''', \mathcal{R}''', K''', \vartheta, \ell \rangle$$
$$\Gamma'_{pc^{\texttt{C.m}}+1} = \Gamma'_{pc^{\texttt{C.m}}} \oplus flat(\vartheta) \quad F'_{pc^{\texttt{C.m}}} = F'_{pc^{\texttt{C.m}}+1} \quad Z'_{pc^{\texttt{C.m}}} = Z'_{pc^{\texttt{C.m}}+1} \quad K'_{pc^{\texttt{C.m}}+1} = K'_{pc^{\texttt{C.m}}}$$
$$\mathcal{T}_{pc^{\texttt{C.m}}+1}, \mathcal{R}_{pc^{\texttt{C.m}}+1} = \begin{cases} \mathcal{T}_{pc^{\texttt{C.m}}} \cup \{o\} \cup \mathcal{T}''', \ \mathcal{R}_{pc^{\texttt{C.m}}} \cup \mathcal{R}''' & \text{if } i \text{ is not recursive} \\ & \text{and } o \notin \mathcal{T}_{pc^{\texttt{C.m}}} \cup \mathcal{R}_{pc^{\texttt{C.m}}} \\ \mathcal{T}_{pc^{\texttt{C.m}}}, \ \mathcal{R}_{pc^{\texttt{C.m}}} \cup \mathcal{R}''' \cup \mathcal{T}''' \cup \{o\} & \text{otherwise} \end{cases}$$

Let $S'_{pc^{\texttt{C.m}}} \approx o \cdot s_1$; then $S'_{pc^{\texttt{C.m}}+1} \approx s_1$. Therefore we obtain that

$$P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash \langle (pc^{\texttt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t : \ell_{pc^{\texttt{C.m}}+1} \tag{4.5}$$

where $\ell_{pc^{\texttt{C.m}}+1} = (\sum \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}}+1)} + \ell_\varphi$. By Lemma 4.9.2,

$$\Big(\text{BCT}, \Gamma^{\texttt{D.run}}, F^{\texttt{D.run}}, S^{\texttt{D.run}}, Z^{\texttt{D.run}}, T^{\texttt{D.run}}, K^{\texttt{D.run}}, i \vdash_t P[\texttt{D.run}] : \ell_i^{\texttt{D.run}} \,;\, \Psi_i\Big)^{i \in reachable(P[\texttt{D.run}], 1^{\texttt{D.run}})},$$

where

$F_1^{\texttt{D.run}} = F_\top^{\texttt{D.run}}[0 \mapsto root(\tau)]$,
$\Gamma_1^{\texttt{D.run}} = flat(\tau)$,
$S_1^{\texttt{D.run}} = \varepsilon$,
$Z_1^{\texttt{D.run}} = \lceil Z'_{pc^{\texttt{C.m}}} \rceil$,
$\mathcal{T}_1^{\texttt{D.run}} = \varnothing$,
$\mathcal{R}_1^{\texttt{D.run}} = \varnothing$,
$K_1^{\texttt{D.run}} = \varnothing$,
$\ell^{\texttt{D.run}} = \sum_{i \in reachable(P[\texttt{D.run}], 1^{\texttt{D.run}})} \ell_i^{\texttt{D.run}}$,
$\bigsqcup_{i \in reachable(P[\texttt{D.run}], 1^{\texttt{D.run}})} \Psi_i = (\vartheta, \lceil Z'_{pc^{\texttt{C.m}}} \rceil, \mathcal{T}'^{\texttt{D.run}}, \mathcal{R}'^{\texttt{D.run}}, K'^{\texttt{D.run}}, \Gamma'^{\texttt{D.run}})$,
$\vartheta = mk\_tree(\Gamma'^{\texttt{D.run}}, o)$,
$\bar{b} = fn(void, \mathcal{T}'^{\texttt{D.run}}, \mathcal{R}'^{\texttt{D.run}}, K'^{\texttt{D.run}}, \vartheta) \setminus fn(\tau, o, lock_o)$.

Thus

$$P, \text{BCT}, \Gamma^{\texttt{D.run}} \cup \Gamma, F^{\texttt{D.run}} \cdot F, S^{\texttt{D.run}} \cdot S, Z^{\texttt{D.run}} \cdot Z, T^{\texttt{D.run}} \cup \mathcal{T}, \mathcal{R}, K^{\texttt{D.run}} \cup K \vdash \\ \big(\Vdash_H \langle (pc^{\texttt{C.m}} + 1, f_1, s_1) \cdot \varphi, z \rangle_t \ , \\ \langle (1^{\texttt{D.run}}, f_2[0 \mapsto o], \epsilon), \epsilon \rangle_o) : \ell^{\texttt{D.run}} \,\&\, \ell_{pc^{\texttt{C.m}}+1}.$$

and $\ell \geqslant \ell^{\texttt{D.run}} \,\&\, \ell_{pc^{\texttt{C.m}}+1}$

The last step is to prove that $\Gamma^{\texttt{D.run}} \cup \Gamma \approx H$, which follows directly from Definition 9.3 and the fact that $\Gamma \approx H$.

Case `return`. We have

$$P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\texttt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\texttt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t) : \ell$$

The following rules may have been applied: (S-RETURN), (S-RETURN-RUN), (S-RETURN-SYNCH-0), and (S-RETURN-SYNCH-N). Let us consider the first one (the other ones are similar).

$$\text{(S-RETURN)} \quad \frac{P[pc^{\texttt{C.m}}] = \texttt{return} \quad \texttt{synchronized} \notin mod(\texttt{C.m})}{P \Vdash_H \langle (pc^{\texttt{C.m}}, f_1, v \cdot s_1) \cdot (pc^{\texttt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t \rightarrow \Vdash_H \langle pc^{\texttt{D.m}}, f_2, v \cdot s_2) \cdot \varphi, z \rangle_t}$$

By the configuration typing rules, we have

$$\begin{gathered} (\text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K', i \vdash_t P[\texttt{C.m}] : \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})} \\ P, \text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'' \vdash \langle (pc^{\texttt{D.m}}, f_2, \bullet \cdot s_2) \cdot \varphi, z \rangle_t : \ell_\varphi \quad \Gamma = \Gamma' \cup \Gamma'' \\ F = F' \cdot F'' \quad S = S' \cdot S'' \quad Z = Z' \cdot Z'' \quad \mathcal{T} = \mathcal{T}' \cup \mathcal{T}'' \quad \mathcal{R} = \mathcal{R}' \cup \mathcal{R}'' \quad K = K' \cup K'' \\ Z'_{pc^{\texttt{C.m}}} \approx z \quad S'_{pc^{\texttt{C.m}}} \approx v \cdot s_1 \quad F'_{pc^{\texttt{C.m}}} \approx f_1 \end{gathered}$$

Therefore

$$\ell = ((\sum \ell_i)^{i \in reachable(P[\texttt{C.m}], pc^{\texttt{C.m}})} + (\sum \ell_i)^{i \in reachable(P[\texttt{D.m}], pc^{\texttt{D.m}})} + \ell_\varphi) \qquad (4.6)$$

Moreover, $S''_{pc^{\texttt{D.m}'}} \approx \bullet \cdot s_2$ and $S''_{pc^{\texttt{D.m}'}}$ has been set while typing the invocation of `C.m` as $S''_{pc^{\texttt{D.m}'}} = root(\vartheta) \cdot S'''$, where $root(\theta)$ is the value returned by the method invocation, then $S''_{pc^{\texttt{D.m}'}} \approx v \cdot s_2$.

Therefore we obtain

$$\ell' = (\sum \ell_i)^{i \in reachable(P[\texttt{D.m}], pc^{\texttt{D.m}})} + \ell_\varphi \qquad (4.7)$$

then $\ell \geqslant \ell'$.

Case `new`. We have

$$P, \text{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle (pc^{\texttt{C.m}}, f, s) \cdot \varphi, z \rangle_t) : \ell$$

and

$$\frac{P[pc^{\mathtt{C.m}}] = \mathtt{new\ D} \quad o \notin dom(H) \quad H' = H[o \mapsto (\rho^D, D)]}{P \Vdash_H \langle (pc^{\mathtt{C.m}}, f, s) \cdot \varphi, z \rangle_t \to \Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z \rangle_t} \text{(S-NEW)}$$

By the configuration typing rules, we get

$$\begin{gathered}(\text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} \\ P, \text{BCT}, \Gamma''', F''', S''', Z''', \mathcal{T}''', \mathcal{R}''', K''' \vdash \langle \varphi, z \backslash Z'' \rangle_t : \ell_\varphi \\ \Gamma = \Gamma'' \cup \Gamma''' \quad F = F'' \cdot F''' \quad S = S'' \cdot S''' \\ Z = Z'' \cdot Z''' \quad \mathcal{T} = \mathcal{T}'' \cup \mathcal{T}''' \quad K = K'' \cup K''' \\ Z \approx z \quad S''_{pc^{\mathtt{C.m}}} \approx s \quad F''_{pc^{\mathtt{C.m}}} \approx f\end{gathered}$$

Therefore

$$\ell = ((\sum \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})} + \ell_\varphi) \tag{4.8}$$

From $(\text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}})}$ and Definition 9.5 we derive that

$$\text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'', pc^{\mathtt{C.m}} \vdash_t P[\mathtt{C.m}] : \ell_{pc^{\mathtt{C.m}}}$$

and

$$(\text{BCT}, \Gamma'', F'', S'', Z'', \mathcal{T}'', \mathcal{R}'', K'', i \vdash_t P[\mathtt{C.m}] : \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)}.$$

Therefore $\ell = (\ell_{pc^{\mathtt{C.m}}} + (\sum \ell_i)^{i \in reachable(P[\mathtt{C.m}], pc^{\mathtt{C.m}}+1)}) + \ell_\varphi$.

By these hypotheses, it is easy to show the existence of Γ', F', S', Z', $\mathcal{T}'$, $\mathcal{R}'$, K' and ℓ' such that $\Gamma' \approx H'$ and $P, \text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K' \vdash (\Vdash_{H'} \langle (pc^{\mathtt{C.m}} + 1, f, o \cdot s) \cdot \varphi, z \rangle_t) : \ell'$. In particular, letting $\mathsf{fields}(\mathtt{D}) = \overline{\mathtt{f}}$, $\Gamma' = \Gamma[o \mapsto ([\overline{\mathtt{f} : \top}], \mathtt{D})]$ and $S'_{pc^{\mathtt{C.m}}+1} = o \cdot S''_{pc^{\mathtt{C.m}}}$.

Here there is a critical point of the proof of subject reduction because it is not evident that $\ell \geqslant \ell'$. Actually, by `new` typing rule $a = names(pc^{\mathtt{C.m}})$, $\Gamma''_{pc^{\mathtt{C.m}}+1} = \Gamma''_{pc^{\mathtt{C.m}}}[a \mapsto ([\overline{\mathtt{f} : T}], \mathtt{D})]$, and $S''_{pc^{\mathtt{C.m}}+1} = a \cdot S''_{pc^{\mathtt{C.m}}}$ and we cannot assume that $a = o$. In particular, since $a = names(pc^{\mathtt{C.m}})$, it may be a name that is already "in use" – this is the case when `C.m` is either recursive or $pc^{\mathtt{C.m}}$ is inside an iteration.

There are two cases: (i) `D` is not a subclass of `Thread` and (ii) `D` is a subclass of `Thread`. In case (i) it easy to verify that $\ell \geqslant \ell'\{^a/_o\} \geqslant \ell'$. In case (ii) we may have $\ell' = (b, c)_a \ \& \ (c, b)_o$. When ℓ' has such a value, we have $\ell'\{^a/_o\} \not\geqslant \ell'$ because the name o may appear as index of a dependency pair. In such situation, $(b, c)_a \ \& \ (c, b)_o$ is a circular dependency, while $\ell'\{^a/_o\} = (b, c)_a \ \& \ (c, b)_a$ is not – see Sections 4.6 and 4.9.5.

To solve this criticality, whenever a new thread is created – either because a `start` is executed or because it is returned by a method invocation, *cf.* the sets $\mathcal{T}$ and $\mathcal{R}$ – we verify whether we are inside a recursion or an iteration, which are the two cases when $names(pc^{\mathtt{C.m}})$ may return a name that already exists. In these two cases, the name a is added to the set $\mathcal{R}$ instead of $\mathcal{T}$. As a consequence, the lam ℓ' will contain terms $\mathtt{RUN}((o[\![\mathtt{f} : \tau]\!], \mathtt{D}))$ while, correspondingly, ℓ will contain terms $\mathtt{RUN}((o[\![\mathtt{f} : \tau]\!], \mathtt{D}))\{^{a}/_{o}\}$. However, since

$$\mathtt{RUN}((o[\![\mathtt{f} : \tau]\!], \mathtt{D})) = \ \mathtt{D.run}((o[\![\mathtt{f} : \tau]\!], \mathtt{D}), o, lock_o) \ \& \ (\nu\, o')\, \mathtt{RUN}((o'[\![\mathtt{f} : \tau]\!], \mathtt{D}))\ ,$$

replacing o in $\mathtt{RUN}((o[\![\mathtt{f} : \tau]\!], \mathtt{D}))$ with any other object name does not change the circularity predicate (because the function is unfolded in a lam that has an invocation of the same function on a fresh name in parallel). Therefore, it is possible to derive $\ell\{^{a}/_{t}\} \succcurlyeq \ell$. □

4.9.8 Deadlocks and circularities

To conclude the proof of correctness we need to bridge the gap between the notion of deadlock in `JVM` and the notion of circularity in a lam program.

Definition 9.6. *A* `JVM` *configuration* $\Vdash_H \langle\varphi_1, z_1\rangle_{t_1} \cdots \langle\varphi_n, z_n\rangle_{t_n}$ *is* deadlocked *if there are* $i_1, \cdots, i_k \in 1..n$ *such that, for every* $j \in \{i_1, \cdots, i_k\}$ *one of the following holds*

- $\varphi_j = (pc_j, f_j, s_j) \cdot \varphi'_j$ *and* $P[\![pc_j]\!] =$ `monitorenter` *and* $s_j = a \cdot s'_j$ *and* $a \in z_h$ *with* $h \in \{i_1, \cdots, i_k\} \backslash j$;
- $\varphi_j = (pc_j, f_j, s_j) \cdot \varphi'_j$ *and* $P[\![pc_j]\!] =$ `join` *and* $s_j = t_h \cdot s'_j$ *with* $h \in \{i_1, \cdots, i_k\}$.

The following statement is a straightforward consequence of the definitions.

Proposition 4.9.4. *Let* $\Vdash_H \mathfrak{C}$ *be deadlocked and let* $P, \mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \mathfrak{C}) : \ell$. *Then* ℓ *has a circularity.*

Theorem 4.9.5. *Let* P *be a* $\mathtt{JVML}_d$ *program and* $\Gamma \approx H$, $F \approx_\Gamma f_\perp[\![0 \mapsto main]\!]$, $S \approx_\Gamma \epsilon$, *and* $Z \approx_\Gamma \epsilon$. *If*

1. $P, \mathrm{BCT}, \Gamma, F, S, Z, \mathcal{T}, \mathcal{R}, K \vdash (\Vdash_H \langle(1^{\mathtt{C}.main}, f_\perp[\![0 \mapsto main]\!], \epsilon), \epsilon\rangle_{main}) : \ell$
2. *and* $\Vdash_H \langle(1^{\mathtt{C}.main}, f_\perp[\![0 \mapsto main]\!], \epsilon), \epsilon\rangle_{main} \rightarrow^* \Vdash_{H'} \mathfrak{C}'$
3. *and* $\Vdash_{H'} \mathfrak{C}'$ *is deadlocked*

then ℓ has a circularity.

Proof. The proof is an immediate consequence of the previous results. By 1 and 2 and Theorem 4.9.3 we have the existence of ℓ' and Γ', F', S', Z', $\mathcal{T}'$,$\mathcal{R}'$,K' such that $\Gamma' \approx H'$ and $P, \text{BCT}, \Gamma', F', S', Z', \mathcal{T}', \mathcal{R}', K' \vdash (\Vdash_{H'} \mathfrak{C}') : \ell'$ and $\ell \geqslant \ell'$. By 3 and Proposition 4.9.4, we have that ℓ' has a circularity. By $\ell \geqslant \ell'$ and Lemma 4.9.1, we have that ℓ has a circularity as well. □

4.10 Related work

Several techniques have been designed for detecting deadlocks of `Java` programs. We start by briefly discussing a set of techniques addressing the analysis of `Java` bytecode. We then discuss separately deadlock analysis techniques using static-time techniques and those requiring either model generation or runtime executions.

4.10.1 Static approaches

Analyses using static time techniques are grouped according to the approach used.

Type Systems

There have been few works based on type systems applied specifically to the deadlock detection in Java. Some of the approaches discussed below do not target this problem in particular, however their techniques, to some extent, tribute to solve some of the problems faced in the development of our solution.

A work that has been key for the development of our solution is [49]. The target of the analysis in this work was not precisely the deadlock verification, instead, this work focused on the verification of well-formedness of the `Java` bytecode with respect to the locking primitives. The approach taken in `JaDA` for the design of the behavioral type system and for tackling complex `Java` features like exception handling has been, to some extent, taken from this work.

Another work that has a lot in common with ours is the deadlock analyzer for `ABS` programs [35]. In fact, our solution can be seen as the natural extension of this work to the `Java` language. This work and ours share the same approach based on behavioral type systems, moreover the abstract representations (*lams*) and the subsequent analysis in both works are quite similar. The main differences are introduced according to the necessity of

tackling a diverse concurrency model and also in relation with some of the intrinsic features of the `Java` language.

On the other hand, one of the most complete works regarding deadlock analysis in `Java` with type systems is [19], which defines a type system that derives the order of lock acquisitions in *SafeJava* programs (a subset of `Java` *with annotations* written as part of the code). Well-typed programs will be verified deadlock free. An extension of this type system for `JVML` has been defined in [54].

Our technique differs from this approach in two key aspects. First of all, we aim to a fully automatic tool allowing to analyze existing programs without user intervention. In addition, our types are behavioral. Therefore, deadlock-freedom is not a property of the type system, instead the deadlock detection is performed over the behaviors obtained by the typing process.

Another example of a type system based approach is the work from [25] for detecting race conditions. Like our proposal the type system annotates the access to shared variables and detect read/write contradictions between different threads that may occur concurrently. The downside of this approach is the little support for object-oriented concepts like complex data-types and inheritance.

Data Flow Analysis

A standard approach for the detection of circular dependencies in concurrent programs is based on the construction of an execution flow graph and then search for cycles within this graph. Due to aliasing – two process may access the same shared resource by using a different names –, this construction quickly becomes complex because elements with different names in the graph may refer to the same object. Tools like `Jlint` use data-flow analysis to deal with aliasing (`Jlint` can be downloaded at `http://jlint.sourceforge.net/`) [12].

Our tool uses an special names generation function for coping with aliases; we also rely on data flow analysis for determining more precisely type of method bodies and method signatures.

Another work using data flow analysis for coping with aliasing is [59]. This technique defines a formal set of rules that control the lock order in their data flow analysis, in similar way to the type system proposed in this thesis. Like our technique this work presents a sound strategy for detecting re-entrant locks, however unlike ours the re-entrance is only detected when the lock expressions involve local variables but not fields. Also, unlike our tool, this technique does not detect circularities with a common prefix – so false positives –, thus leading to a higher number of false positive outcomes.

The work of Nayik [51] defines a set of necessary conditions for the existence of deadlocks, which is then verified by data flow analysis. This is the theory behind the `Chord` tool discussed in Section 5.4. We notice that this tool is reported as one of the most effective ones by [26, 51].

Another technique based on this approach is the work of [27]. This technique uses a points-to analysis in combination with a *may happen in parallel* analysis to detect circular dependencies in the code. This technique does not target the `Java` language directly, instead it targets actor-based concurrency models for which there are some implementations of third party concurrency libraries for `Java`. Like this technique our solution, ensures that the points of the program involved in dependency cycles, can indeed happen concurrently. This analysis ensures the removal of a considerable number of false positives.

4.10.2 Non-static approaches

Non static techniques usually analyze programs that either have finite models or that have a finite set of relevant inputs. The advantage of these techniques is that it is possible reduce the case explosion and obtain a very precise analysis.

Model Checking

A model checking approach composed of two steps is presented by [47]. A first step generates a so called *trace program* that records the critical concurrent operations and discards non critical parts. In a second step this program is analyzed using an off-the-shelf model checker. Like this technique, our approach tries to get rid of non critical parts of the program. In our case, we use lams, thus allowing to perform the analysis in a reduced but key part of the program, without constraining the program model to be finite.

Monitoring Analysis

Monitoring techniques allow detecting potential deadlocks at runtime. The main idea behind this approach is to analyze only real scenarios. The analysis of circularities in our approach has some similarities to the one presented in [17]. In particular, like this approach, in our one, the locked objects are tagged with the label identifying the threads acquiring the locks. This allows to deal with re-entrance problems in a very clever and simple way. On the other hand the two approaches differ substantially when parallel codes must be detected. Our technique detects the parallel states in the type system,

which is done at static time and, therefore, some over-approximation is introduced to cope with the non-determinism of the scheduler. In [17], being the analysis at runtime, it is possible to tag each segment of the program that is reached by the execution flow, in a manner that these tags specify the exact order of lock acquisitions. The technique in [17] is also interesting because it uses an hybrid strategy for detecting potential deadlocks that might occur because of different scheduler choices (than the current one). The theory of [17] has been prototyped in the `GoodLock` tool used in our benchmark results comparison in Section 5.4. This technique has been refined in `Sherlock`, a tool using symbolic executions [26].

4.11 Conclusions

Deadlocks have proven to be one of the most common flaws in modern concurrent software. Only in the *Oracle Bug Database*, the issue tracking system of the Java Development Kit (`JDK`), there are currently more than 40 unresolved deadlock problems.

In this chapter we have presented the theoretical tools for the analysis of deadlocks in `Java` programs. Our approach is based on a behavioral type system for `JVML` (the `Java` bytecode) that abstracts object dependencies in concurrent programs. These dependencies are then analyzed by means of a decision algorithm that detects circularities.

There are two features in the `Java` language that entangle the most the analysis process: the recursive invocations and the creation of fresh objects (that are not part of the original arguments), and in particular, the creation of new threads. The combination of these two features may lead to models with potentially infinite dependencies. The strength of the analysis here proposed is the ability to reduce these models to finite ones.

Existing techniques for the analysis of deadlocks in `Java` programs are either unsound [14, 51] or target only a subset of the language [54]. In our solution we put special attention to both of these aspects.

We have successfully managed to analyze most of the complex features of `Java` like: recursive data types, static members, inheritance, arrays structures and exception handling. We have also provided a formal proof of the correctness of the type system and the ulterior analysis.

The correctness of this approach is proven in two parts. The first part focuses in the correctness of the type system. This proof is, as usual, done by demonstrating that the subject reduction theorem holds for the static semantics of the type system. The second part of the proof focuses on demonstrating that the type (and therefore the dependencies) extracted at static

time is preserved during the runtime execution of the program.

Still, for some `Java` programs this typing process may not succeed, static deadlock analysis is after all, undecidable. Our type system may fail, in particular, in presence of data race problems, that do not allow identifying the exact behavior of the program.

There are two major elements of the `Java` concurrency features that remained out of the scope of this analysis: the synchronization mechanisms based on the `wait`, `notify` and `notifyAll` methods; and the components of the `java.util.concurrent` package.

The first of these features has been discussed in this chapter. We noted that the analysis of circularities involving these methods can be dual to the analysis targeting the synchronize blocks constructions treated in our solution. We remark that the modularity of our solution could easily allow to extend the analysis process by adding the new verifications like the one proposed to this aim.

Chapter 5

The JaDA tool

Summary

In this chapter we describe the JaDA Tool. This is the prototype implementation of the theory discussed in Chapter 4. We describe in depth the algorithms for the type system inference and the behavioral types analysis. We also review the existing works in the literature addressing these or similar problems. We assess the performance of JaDA against some of these, as well as, against a commercial grade tool. These assessments involve a set of well known and ad-hoc Java and Scala programs. The chapter ends with some final remarks about this solution.

5.1 Introduction

The proposed solution follows the general approach described in Chapter 1. Figure 5.1 particularizes this approach for the case of deadlock analysis in Java programs.

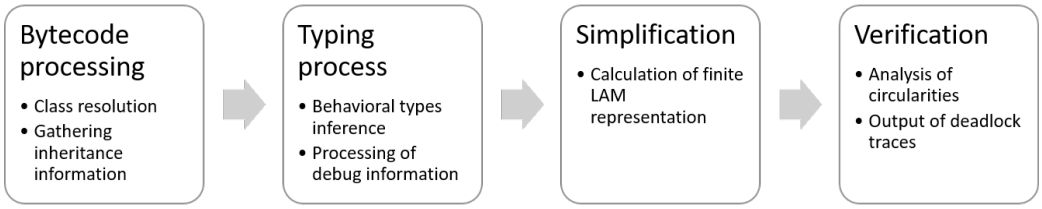

Figure 5.1: Behavioral types based approach for Java deadlock analysis

For the typing process, we have defined an inference system that associates abstract behaviors to Java bytecode instructions. This system consists

of a number of rules which are identical for most of the instructions except for those that have effects in the synchronization process. The rules that need to be treated in a special way are invocations, locks acquisitions and releases, object manipulation and control flow operations. The abstract behaviors extracted from Java bytecode, called *lams* [34, 48], are infinite state models that define dependencies between threads. The authors of [34, 48] demonstrated that the deadlock analysis in lams is decidable and proposed an algorithm to this purpose. In this work we extend both the lams model and the deadlock detection algorithm in order to cope with some of the Java features.

The inference of the behavioral types and this algorithm constitute the core of a prototype called JaDA – the Java *Deadlock Analyzer* –. The current release of JaDA handles all the main features of Java including threads and synchronizations, constructors, arrays, exceptions, static members, interfaces, inheritance, recursive data types. Few features are not yet covered in the current JaDA version, i.e., wait-notify-notifyAll operations, and constructs involving the Java volatile modifier.

Chapter contents

The contents of this Chapter correspond to the description of JaDA: the prototype implementation of the theory presented in Chapter 4. Sections 5.2 and 5.3 present the core algorithms behind this tool. The former corresponds to the automatic inference of the behavioral types out of bytecode instructions, while the latter corresponds to the analysis of the circularities within these behaviors. In Section 5.4 we compare the results of JaDA against the results of some of the existing tools, including a commercial grade one. Section 5.5 describes the main features of the tool and its configuration possibilities. Finally, this chapter concludes in Section 5.6 with the main remarks about the JaDA tool, and the possible future lines of work based upon this solution.

5.2 Type inference

The theory described in Chapter 4 relies on program annotations (see rules for method definitions in Figure 4.5 and the function BCT). However, the JaDA tools does not require any of these annotation to be made by the programmer. It is able to *infer* types out of the code. The implementation of the type inference mechanism and the further analysis of the resulting behavioral use two iterative processes.

The inference of lams and method types from JVML programs is an iterative process that, given a set of method types (which include method effects, such as the structure and the updates of the arguments and the returned object, when applicable, the new threads spawned and the synchronized threads), at each step, computes the types of method instructions and the method types. The overall process reiterates again if method types are modified, till an invariant is reached.

In order to infer the right types we use a standard solution that relies on subtypes and relaxes the equality constraint of types of successor instructions in Figures 4.4 and 4.5. We briefly outline this solution.

Record types ϑ are equipped with a binary relation $<:$ that is the reflexive and transitive closure of the following pairs:

- $\vartheta <: \top$, and
- $(a[\cdots, \mathtt{f}^{\mathtt{h}} : \vartheta, \cdots], \mathtt{C}) <: (a[\cdots, \mathtt{f}^{\mathtt{h}'} : \vartheta', \cdots], \mathtt{C}')$, if $\mathtt{C} \subseteq \mathtt{C}'$ and, for all fields, $\mathtt{h} \leqslant \mathtt{h}'$ and $\vartheta <: \vartheta'$.

(we also have $\bot <: \vartheta$, where $\bot$ has been introduced to deal with inheritance, see Section 4.8). We notice that the constraint $\mathtt{C} \subseteq \mathtt{C}'$ is the identity in $\mathtt{JVML}_d$, but it is subset relation when inheritance comes into the scene. Clearly, $<:$ defines an upper-semi lattice on record types, which allows us to use the operation $\sqcup$ for computing least upper-bounds in our algorithm. The relation $<:$ is extended to functions Γ_i, F_i and S_i as follows. Let $\Gamma_i <: \Gamma_i{}'$ whenever Γ_i and $\Gamma_i{}'$ have the same domain and, for every χ, $mk\_tree(\Gamma_i, \chi) <: mk\_tree(\Gamma_i{}', \chi)$. Similarly for $F_i <: F_i{}'$, with the assumption that we have an implicit environment Γ_i for computing the record types of $F_i(x)$ and $F_i{}'(x)$. As regards stacks, $S_i <: S_i{}'$ whenever $S_i = \chi_1 \cdots \chi_n$, $S_i{}' = \chi'_1 \cdots \chi'_n$ (they have the same length) and, for every j, $\Gamma_i(\chi_j) <: \Gamma_i(\chi'_j)$. Therefore, in the typing rules of Figures 4.4 and 4.5, the equality constraints between Γ_i, F_i, and S_i and Γ_{i+1}, F_{i+1}, and S_{i+1} are relaxed in favor of $<:$. For instance $\Gamma_i = \Gamma_{i+1}$ becomes $\Gamma_i <: \Gamma_{i+1}$. We also relax identities $T_i = T_{i+1}$ into $T_i \subseteq T_{i+1}$.

The process typing the instructions uses a data-flow algorithm whose fix point is reached when all constraints are satisfied. The pseudo-code of the algorithm is the following (a visual diagram of the algorithm is shown in Figure 5.2):

```
1  TypeInference(Bct, P, Env_0) -> <Env, L>
2      Env = array[P.size]
3      L = array[P.size]
4      Env[0] = Env_0
5      pending = {0}
6      while( !pending.isEmpty() )
7          index = pending.pop()
8          current = P[index]
9          L[index] = current.type.lam(Bct,Env[index])
10         next_env = current.type.env(Bct,Env[index])
11         if( !current.isReturn() )
12             if( Merge(next_env, current.next, Env) )
13                 pending.push(current.next)
14             if( current.isIf() )
15                 if( Merge(next_env, current.jump, Env) )
16                     pending.push(current.jump)
```

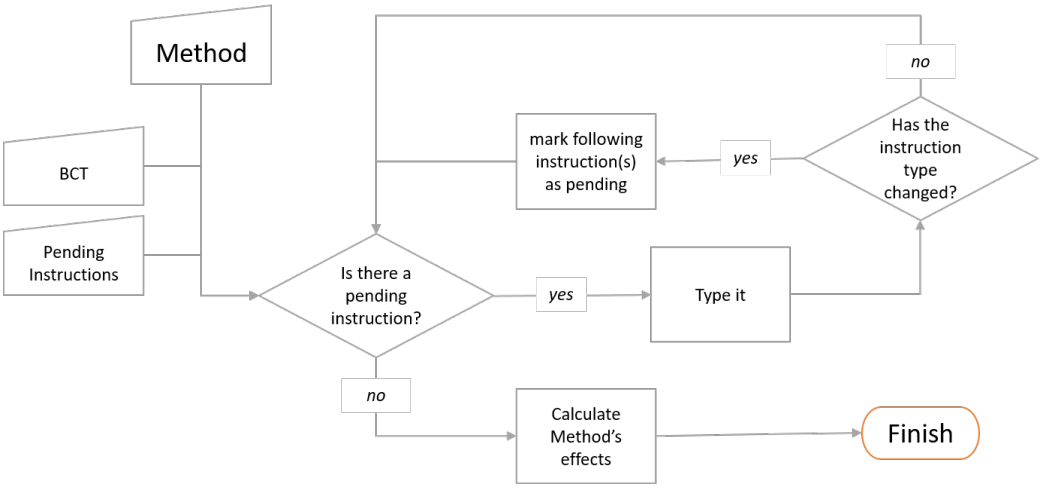

Figure 5.2: Type inference of method's bodies in JaDA.

`TypeInference` starts with the typing environment of the first instruction (see the rules for method definitions in Section 4.5.1) and initializes the set `pending` to $\{0\}$ (only instruction `0` must be typed (see line 5). The typing environment includes Γ, F, S, Z, T, R, and K of Section 4.5. The type of every instruction is saved in `L` (line 9). After typing the current instruction we get the typing environment for the next instruction; this environment has to match the current typing environment for the next instruction (line 12). If the two typing environments do not match, they are merged using the `<:` relation and the corresponding instruction has to be typed again (line 13). Notice that in a first pass all instructions will be typed and their

typing environment computed, this is because the initial environment for every instruction (except the first one) is `null`. If the current instruction is an `if` *jump* then the same process is repeated now comparing with the environment at instruction *jump*. The `Merge` function checks if the existing environment is `null`, if so the environment for the current instruction is updated and the method returns true because the instruction needs to be typed again. If an environment for the next instruction already exists then the existing environment is merged with the one just computed one and the function `Merge` returns true again. `Merge` returns false if the existing typing environment remains unchanged.

The process `TypeInference` terminates because, at every iteration, the environment is augmented or remains unchanged (because of the merge operation). In a method body object names are finite (see paragraph *The function names*$(\cdot)$ in Section 4.6). Our prototype does not use recursive records (recursive types are banned in $\texttt{JVML}_d$ and are represented by finite structure in `JVML`, see Section 4.8). Additionally the upper-semi lattice of `<:` has a finite height. These premises give necessary conditions for the termination of the above process.

The `TypeInference` process assumes the existence of a BCT. Actually, the computation of the BCT is performed by another iterative process that uses `TypeInference`. The following pseudo code highlights the algorithm (a visual representation of the algorithm is shown in Figure 5.3):

```
1  BCT(ClassList) -> <Bct>
2      foreach class in ClassList
3          foreach m in class.Methods
4              Bct[m] = Empty(m)
5
6      changes = true
7      while(changes)
8           changes = false
9           foreach class in ClassList
10              foreach m in class.Methods
11                  <Env, L> = TypeInference(Bct, m.P, m.Env_0)
12                  changes = changes OR Update(Bct, m, Effects(L))
```

`BCT` starts by initializing the effects of every method to be empty (line 4). Then every method body is typed with the current `Bct` (line 11) and the typing environments, lams and the method's effects are re-computed from its type (12). This process is repeated until the `Bct` reaches a stable state (line 7). The arguments for the termination of `Bct` are similar to those of `TypeInference`.

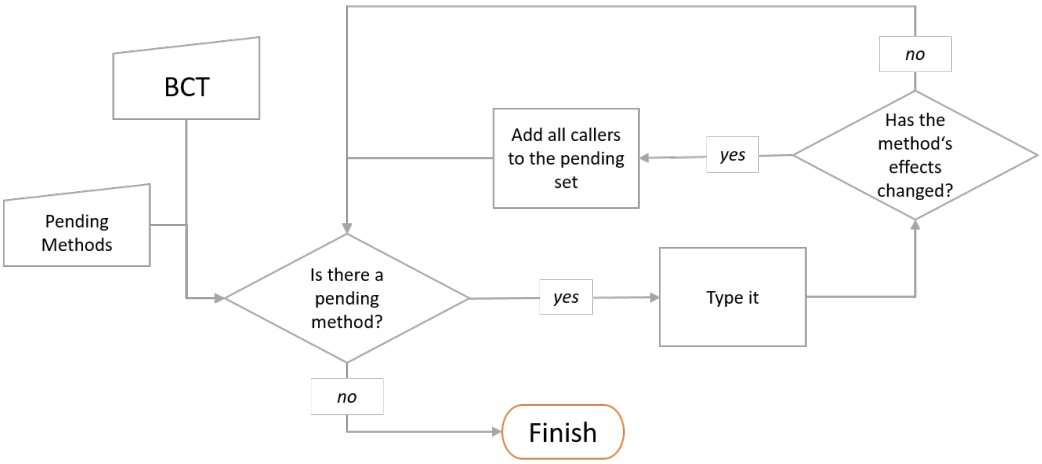

Figure 5.3: Type inference of methods' behaviors in JaDA.

5.3 Analysis of circularities

The pseudo code of the algorithm for computing the abstract model of a lam function in Section 4.6 is highlighted here (a visual representation of the algorithm is shown in Figure 5.4):

```
1  ExpandAndCleanCCT(Bct)
2      foreach (method in Bct)
3          State[method] = {{}}
4          do
5              changed = false
6              foreach( method in Bct )
7                  newState = Replace(State, method.type.lam)
8                  newState = Normalize(newState)
9                  foreach( set in newState )
10                     set = Transitive_Closure(set)
11                     set = Clean(set)
12                 changed = changed OR IsEqual(State[method],newState)
13                 State[method] = newState
14         while( changed )
```

`ExpandAndCleanCCT` computes the set of states of every method in the BCT. For each method, the function computes its state (line 6) by instantiating the method invocations in its lam with the models computed in the previous iteration (line 7; at the beginning the model is empty, see line 3). Then the state is simplified as a disjunction of conjunctions (the `Normalize` invocation at line 8). That is the model is a set of elements that are sets of dependency pairs. Every set is then transitively closed (line 10) and then simplified by

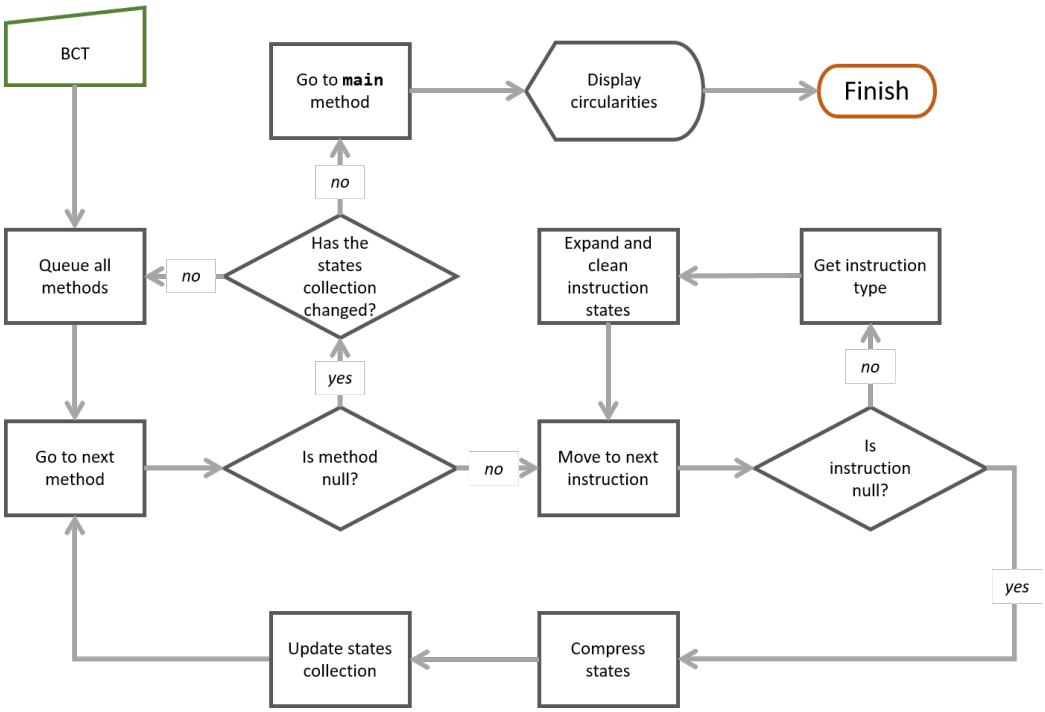

Figure 5.4: JaDA Analysis of behavioral types

means of the Clean operation in line 10 . After all method states have been recomputed, we check if the new model is different than the old one (line 12). This process goes on until every method has reach an stable set of final states.

JaDA returns a deadlock if it finds a circularity in the lam of the main method. Since the method invocations are stored in the structure implementing the dependency pairs, it is also able to return a trace corresponding to the deadlock.

5.4 Related tools and assessments

In this section we compare JaDA with other deadlock analysis tools for Java. For this comparison we have chosen those that, to the bests of our knowledge, are more powerful tools so far. In Table 5.1, the related tools have been classified according to the type of analysis they perform (see Chapter 4.10 for a discussion about analysis techniques for deadlock detection). We have chosen Chord for static analysis [51], GoodLock for hybrid analysis [17] and Sherlock for dynamic analysis [26]. We have also considered a commercial

static analysis tool, ThreadSafe [1] [15].

It is important to remark that we considered in our comparison the Sherlock tool in spite of its dynamic nature. The main reason for this is that Sherlock's results provided a number of *verified* deadlocks for each one of these programs. Therefore, we expect the results of static analyzers to output numbers at least greater than the ones given by Sherlock. We notice that this is not the case either for Chord or ThreadSafe (see example RayTracer).

Table 5.1: Comparison with different deadlock detection tools. The inner cells show the number of deadlocks detected by each tool. The output labeled "*" are imprecise: see the text.

	Static		Hybrid	Dynamic	Commercial
benchmarks	JaDA	Chord	GoodLock	Sherlock	ThreadSafe
Sor	1	1	7	1	4
Hedc	*	24	23	20	1
Vector	*	3	14	4	0
RayTracer	*	1	8	2	0
MolDyn	*	3	6	1	0
MonteCarlo	*	2	23	2	0
Xalan	*	42	210	9	1
BuildNetwork	3	0			0
Philosophers2	1	0			1
PhilosophersN	3	0			0
StaticFields	1	1			1
ThreadArrays	1	1			1
ThreadArraysWJoins	1	1			0
ScalaSimpleDeadlock	1				
ScalaPhilosophersN	3				

The source of all benchmarks in Table 5.1 is available either at [26, 51] or in the JaDA-deadlocks repository[2]. Out of the four chosen tools, we were able to install and effectively test only two of them: Chord and ThreadSafe; the results corresponding to GoodLock and Sherlock come from [26]. We also had problems in testing Chord with some of the examples in the benchmarks, perhaps due to some miss-configurations, that we were not able to solve even after getting in contact with the author because Chord has been discontinued.

The first block belongs to a group of well known Java programs used as benchmarks for several Java analysis tools. In its current state JaDA only detects 1 deadlock (and its full trace) in all of the seven analyzed programs

[1] http://www.contemplateltd.com/threadsafe

[2] https://github.com/abelunibo/Java-Deadlocks

from this group. After a manual review we have noticed that the current release of JaDA does not correctly manage arrays of threads. The tool, though, is able to verify the necessary conditions for deadlock existence, but it is not able to produce the correct traces nor individuate the exact number of deadlocks involved. Enhancing the support for this feature is among the top priorities in the development of JaDA. We have, on the other hand, isolated patterns involving arrays of threads, examples ThreadArrays and ThreadArraysWJoins, and in both case, for this shorter examples the tool outputs the expected results. Moreover, in the case of arrays of threads with join operations, we remark that Chord and ThreadSafe both fail even for a simplified example of this pattern.

In addition, in the case of the Hedc program, the concurrency patterns rely on wait-notify-notifyAll, which are not supported by JaDA.

The other programs in the second block corresponds to examples designed to test our tool against complex deadlock scenarios like the Network program. We notice that both Chord and ThreadSafe fail to detect those kinds of deadlocks. In addition we report two examples of Scala programs [53], these programs have been compiled with the version 2.11 of the Scala compiler. We remark that, to the best of our knowledge at the moment of writing this, there were no static deadlock analysis tools for such language. While the results in Table 5.1 are far from ideal, they are encouraging. We hope to improve more convincing ones as soon as JaDA overcomes its current limitations.

5.5 JaDA Deliverable

JaDA has been designed to run on bytecode generated by the Java compiler[3] and it assumes that the bytecode has been already checked by the Java Bytecode Verifier (therefore it does not contain either syntactic or semantic errors). JaDA also requires that every dependency is matched by a corresponding bytecode. Although the bytecode is not executed, JaDA computes every necessary information to solve key issues for the analysis, such as the informations about inheritance. The loading of the existing types is done dynamically in a sand-boxed class loader[4] to avoid security risks. The full set of dependencies can be specified in JaDA through a *classpath*-like configuration (see property class-path in Section 5.5.1).

JaDA is available in three forms: a demo website [31], a command line tool and an Eclipse plug-in (see Figure 5.5). All of them share the same

[3]We have tested JaDA against the 1.6, 1.7 and 1.8 versions of the Java compiler, and against the 1.8 version of the Eclipse Java Compiler (ECJ)

[4]https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

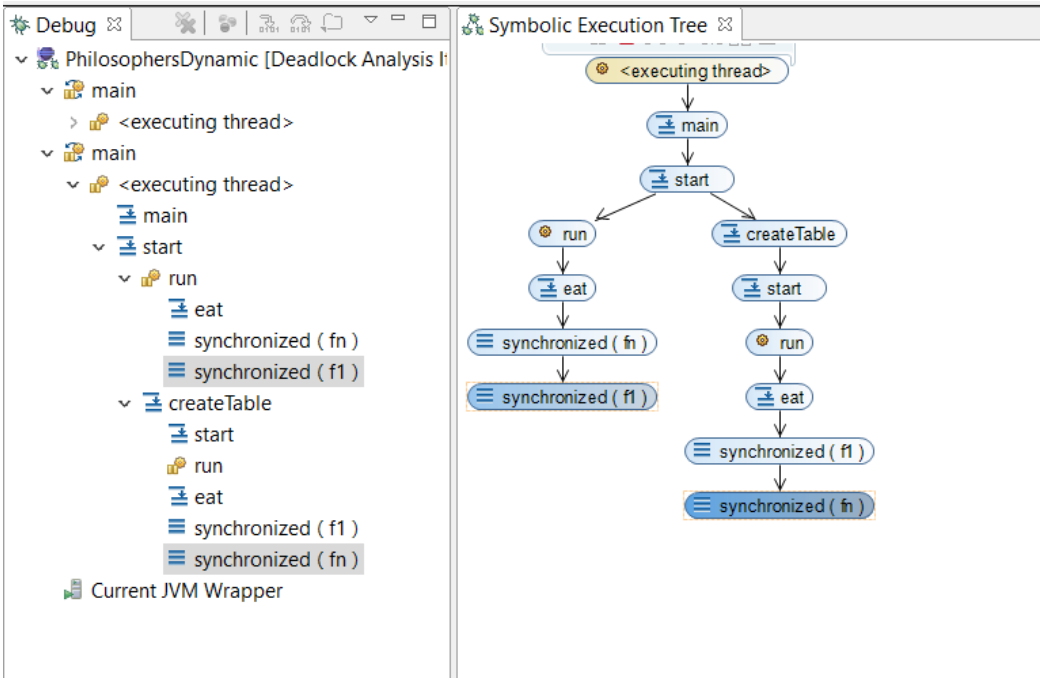

Figure 5.5: Visual representation of a deadlock trace

core: the technique discussed in this work. At the moment of writing, the demo website only allows analyzing single-file programs and to use a subset of the options previously described. The command line tool and the Eclipse plug-in are available through direct requests.

Deadlock trace report

An important result of this solution is the ability to produce the deadlock traces. To this aim the traces of every lock chain have to be maintained at every step of the process. This is allowed by a careful design and some extra work during the type inference process. Since the analysis here presented is static, the resulting traces does not include variable values nor any extra information on the state of the program. It does, however, produce the exact paths involved in the potential deadlock. The Eclipse plug-in, for example, outputs the execution graph causing the deadlock with links to the source code that originates it (see Figure 5.5).

Figure 5.5 shows a deadlock trace representation in the JaDA Eclipse plug-in. This plug-in is the richest delivery form of this prototype, the deadlock trace includes direct links to the source code from where the deadlock is generated. We notice that this is only possible when debugging information

has been compiled into the bytecode.

5.5.1 Customization

The main goal of JaDA is to provide a fully automatic tool for the analysis of deadlocks. We note that even though the user interaction (e.g. through code annotations) may allow gaining in precision, this is a cumbersome task and having a tool completely dependent in this sense may lead to a tool only applicable in non-realistic scenarios.

In order to provide some flexibility, JaDA supports a set of settings to customize the analysis.

<target>: this setting specifies the target file or folder to analyze. It is mandatory. The type of files admitted are: Javaclass files (".class"), Java jar files (".jar") and compressed zip files (".zip"). In the case of folders, the content of the folder is analyzed recursively.

verbose[=<*value*>]: the value ranges from 1 to 5, the default and more expressive value is 5.

class-path <*classpath*>: Standard Java classpath description. If the target contains dependencies other than those in the standard library, they must be specified via this option.

target-method <*methodName*>: fully qualified target method (should be a void method without arguments). It compels JaDA to analyze the specified method. If this option is not set, the analysis chooses the first main method found.

analysis-extent[=<*value*>]: Indicates the extent of the analysis. Possible values are (default value is classpath): full –analyzes every dependency including the system and classpath-included libraries–, classpath –analyzes every library in the classpath–, custom –analyzes the classes specified through the property additional-targets– and self –does not analyze any class but the target one–.

additional-targets <*classes*>: if analysis-extent is set to custom this property must contain a comma separated list of the fully qualified names of a subset of classes in the *classpath* to include in the analysis. Such a feature is useful for avoiding type known libraries.

custom-types <*file*>: a setting file to specify predefined behavioral types.

`static-constructors[=<value>]`: indicates when the static constructors should be processed, the possibilities are `before-all` and `non-deterministically`. The default option is `before-all`.

Third-party tools involved

The analysis process in JaDA starts with the parsing of the bytecode. This is a cumbersome task because of the length and verbosity of the JVML syntax. JaDA relies on the ASM framework [21] for the bytecode extraction and manipulation. (Other third party tools have been also designed for manipulating and analyzing the bytecode: the page `https://java-source.net/open-source/bytecode-libraries` contains a list of existing tools for this purpose. ASM provides a wide set of tools for interacting with the bytecode, including code generation and code analysis. It is also light-weight, open source, very well documented and up to date with the latests versions of Java.

The Eclipse plug-in integration has been inspired in the Symbolic Execution Debugger tool [39].

5.6 Conclusions

In this chapter we have presented the JaDA Tool. It is a static deadlock analysis tool that targets JVML, the Java bytecode. Therefore, it supports the analysis compiled Java programs, as well as, programs written in other languages that are also compiled in JVML, like Scala. In fact, although there are several tools that target Java, this is to the best of our knowledge the first static deadlock analyzer applied to Scala programs.

The technique underlying JaDA uses a behavioral type system that abstracts the main features of the programs with respect to the concurrent operations. These types are then analyzed in order to verify the presence of circular dependencies that can potentially lead to deadlock states. The solution here described has been successfully applied to another programming language: ABS [35]

JaDA is designed to run in a fully automatic fashion, meaning that the inference of the program type and the subsequent analysis could be done unassisted. Nevertheless, user intervention is possible and may enhance the precision of the analysis, for example in presence of native methods. The possibilities to configure JaDA are listed in Section 5.5. In particular, one of the main advantage of this approach is the possibility to provide, manually,

the behavioral types of methods that cannot be typed automatically, like native methods or methods relying on reflection.

Even though the tool is still under development, we have been able to assess it by analyzing a set of `Java` and `Scala` programs. This contribution also reports a comparison between `JaDA`'s results and those of existing deadlock analysis tools, among which is a commercial grade one. The assessments have proved that our tool excels in the analysis of recursive programs that a priori may lead to infinite state models, like the case of the example in Section 4.3.3.

Chapter 6

General conclusions

Summary

We conclude this work with the main remarks about the techniques that has been so far described. We also discuss future lines of research derived from the solutions here presented.

6.1 Conclusions

This thesis describes an approach based on the use of behavioral types for the static analysis of computer programs. We have successfully applied this technique in two different problems, in two different programming languages.

Behavioral type systems constitute a powerful way to reason abstractly in the program effects. The main characteristic of this approach is the modularity, these type systems are built compositionally from the level of program instructions to the level of more general programming units like methods or classes. This poses several advantages, for example, the possibility to focus during the development phase on subsets of the programming language and extend this gradually afterwards to support all the language functionalities. Moreover, this modularity is extended also to the overall system. In this case, the properties targeted by the analysis are not properties of the type system, like it could be, for example, the type safety in standard type systems. Instead, behavioral type systems produce an abstraction of the program that remove spurious information for the analysis. This abstraction can be analyzed afterwards, in a second stage. Having the analysis decoupled from the type system allows to combine several analyzers, but more importantly, eases the demonstration of the correctness of the whole solution.

The demonstration of the correctness of these techniques is usually done

in two steps. A first step, ensures that: (i) the behavior obtained for the program at static time is preserved during the program execution; and (ii) the execution does not introduce at runtime information that was not considered in the program abstraction. This proof is usually done by means of a standard subject reduction theorem demonstrated by induction on the program runtime evolution. In a second part, one can demonstrate the correctness of the analyzer, or even delegate such analysis to other certificated third party tools.

The main contribution of this work is the application of this technique to two problems: the analysis of virtual machines usage in dynamic Cloud Computing scenarios, and the analysis of deadlocks in Java. In both cases we have respectively developed prototype implementations: The `SRA` tool and The `JaDA` tool, that allow to interact with the theory here proposed.

Both tools can be considered novelties in their particular field, at least to some extent. For the case of `SRA`, this is the first automatic analyzer that considers Cloud related resource usage. The language targeted by this analysis is an object-oriented concurrent language that allows the allocation and deallocations of objects representing virtual machines. Moreover, these virtual machines are in turn, active objects, that may run asynchronous operations that also create and release other virtual machine objects. This combination of settings has not been, to the best of our knowledge, considered by any of the existing similar tools. For the case of the `JaDA` tool, this is one of the few analyzers capable of targeting a very wide set of the `Java` language features. It is though, the first of these that proposes a sound approach without impacting in the precision of the analysis. It is also the first analyzer capable of detecting deadlocks in `Scala`, a high end programming language that is also compiled to `Java` bytecode.

6.2 Future work

The results obtained by these two analyzers have been compared with other scientific or commercial tools. In both of the problems tackled, our tools proved competitive, state-of-the-art level results. However, this experience shows that there is still room for improvement, which leaves us with some concrete future lines of work:

`SRA` tool enhancements

Removing technical restrictions. The restrictions described in Section 2.4 are technical simplifications and do not constitute particular limitations to

the theory behind. However, these can be alleviated by retaining more expressive notations for the effect of a method, *i.e.* by considering the methods' possible releases as a set of sets instead of a simple set. Such a notation is more suited for modeling nondeterministic behaviors and it might be made even more expressive by tagging the methods' side effects with conditions specifying when each effect is yielded. Clearly, the management of these domains becomes more complex and the trade-off between simplicity and expressiveness of the behavioral types must be carefully evaluated.

Integrating new automatic solvers. The existing cost equations solvers tend to specialize in particular scenarios. A good alternative to improve the `SRA` tool could be to integrate other (more powerful) solvers, such as those based on theorem provers or in multivariate cost analysis. This extension could definitely enrich the domain of programming patterns that can be covered by this analyzer.

Targeting higher-end languages. A big limitation of the `SRA` tool is the targeted language. This reduces the possibility to analyze *real life* programs. Targeting high end programming languages like `Java` or `C` (as the `C4B` analyzer) would give the `SRA` tool a higher practical value. Another, good step in this direction is to target a programming language with a formal model as `ABS` [46], of which `vml` is a very basic sub-calculus.

In particular, the latter of these lines of work may be of special interest. Behavioral types are abstract specifications of programs that highlight resource usages. One of the main applications of these abstract specifications is to bridge the gap between programs and performance constraints in Service Level Agreement (SLA, in short) documents [32]. As discussed in [32], a language like `ABS` is intended to be a language able to interact, for example, with the Amazon Elastic Cloud Computing features library. Therefore, this language might be used to model Cloud services, where a primary issue is the compatibility with the SLA document. The technique conveyed here would allow (*i*) extracting the abstract specifications by means of the type (inference) system, (*ii*) computing the cost by means of the solver, and (*iii*) verifying whether the computed costs complies with the SLA or not. This solution demonstrates that these steps are all feasible and correct.

`JaDA` tool enhancements

Full `Java` support. A logical first step in this direction is to extend `JaDA` to cover the remaining `Java` features with respect to the concurrency patterns.

As described so far, there are some limitations on the programs that can be analyzed by this tool. The main ones are: the analysis of the high level concurrency support introduced with the `java.util.concurrent` package of the `Java` standard library; the analysis of the synchronization mechanisms provided by the methods `wait`, `notify` and `notifyAll` of the class `Object`; and the analysis of the instructions involving the `Java volatile` keyword.

Enhance the data race problems detection. In addition to deadlocks, another issue closely related to concurrent programming is the data race problem. The behavioral type here described currently keeps track of the access to shared objects from different threads, see *Record Types* in Section 4.4. At this moment, when such scenario is detected, the typing process does not succeed. The `JaDA` tool reports an inconclusive deadlock to ensure the soundness of the output. However, a much more informative output would report the details of the data race condition.

Applying this approach in other high end languages. One of the main contributions of this work has been, without doubts, the application of this technique to a high-end language like `Java`. In this case, the tool targets specifically the `Java` bytecode, `JVML`, which allows analyzing also another sophisticated language like `Scala`. Like `Java`, many common programming languages compile to intermediate stack based semantics bytecode. The technique described in this work for the type inference in `JVML` might be, with more or less effort, adapted to match some of these languages. A good first choice in this sense would be the `IL` language which is the compilation target of languages in the `.Net` platform like `C#, VB, F#`. We remark that thanks to the modularity of this approach, the analysis here described can be re-used without any modifications.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of java programs. In *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, pages 55–71, 2014.

[2] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-sensitive cost analysis for concurrent objects. *Softw. Test. Verif. Reliab.*, 25(3):218–271, May 2015.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings SAS 2008*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 157–172, 2007.

[5] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, pages 113–132, 2007.

[6] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[7] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Peak cost analysis of distributed systems. In *Proceedings of SAS 2014*, volume 8723 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014.

[8] Elvira Albert, Jesús Correas, and Guillermo Román-Díez. Non-Cumulative Resource Analysis. In *Proceedings of TACAS 2015*, Lecture Notes in Computer Science, pages 85–100. Springer, 2015.

[9] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. *SIGPLAN Not.*, 45(8):121–130, 2010.

[10] Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Proceedings of SAS 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2012.

[11] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

[12] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75, 2001.

[13] Robert Atkey. *Amortised Resource Analysis with Separation Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Robert Atkey and Donald Sannella. Threadsafe: Static analysis for java concurrency. *ECEASST*, 72, 2015.

[15] Robert Atkey and Donald Sannella. Threadsafe: Static analysis for java concurrency. *ECEASST*, 72, 2015.

[16] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *Proceedings of LCPC 2005*,

volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2006.

[17] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *in Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2005.

[18] Leonard Berman. The complexity of logical theories. *Theoretical Computer Science*, 11(1):71 – 77, 1980.

[19] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA 2002*, pages 211–230. ACM, 2002.

[20] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Proceedings of TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[21] Eric Bruneton. Asm 4.0 a java bytecode engineering library. `http://download.forge.objectweb.org/asm/asm4-guide.pdf`. Last accessed: 2016-12-03.

[22] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th PLDI Conference*, pages 467–478. ACM, 2015.

[23] Wei-ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for oo programs. In *Proceedings of SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005.

[24] Haskell B Curry, Robert Feys, and William Craig. Combinatory logic. I, 1958. North-Holland, Amsterdam.

[25] Mohamed A El-Zawawy and Hamada A Nayel. Type systems based data race detector. *Computer and Information Science*, 5(4):53, 2012.

[26] Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pages 353–365. ACM, 2014.

[27] Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, pages 273–288, 2013.

[28] Antonio Flores Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of 12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[29] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

[30] Abel Garcia and Cosimo Laneve. Static analyzer of resource usage upper bounds, 2015.

[31] Abel Garcia and Cosimo Laneve. `JaDA` – the `Java` Deadlock Analyzer. Available at `JaDA.cs.unibo.it` from October 28, 2016.

[32] Elena Giachino, Stijn de Gouw, Cosimo Laneve, and Behrooz Nobakht. Statically and dynamically verifiable sla metrics, 2016. To appear on Lecture Notes in Computer Science.

[33] Elena Giachino, Einar Broch Johnsen, Cosimo Laneve, and Ka I Pun. Time complexity of concurrent programs - - A technique based on behavioural types -. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 199–216, 2015.

[34] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.

[35] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in ABS. *Software and Systems Modeling*, 2015.

[36] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[37] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *ACM SIGPLAN Notices*, volume 44, pages 127–139. ACM, 2009.

[38] Arie Gurfinkel and Sagar Chaki. Combining predicate and numeric abstraction for software model checking. *STTT*, 12(6):409–427, 2010.

[39] Martin Hentschel, Richard Bubel, and Reiner Hähnle. Symbolic execution debugger (SED). In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 255–262, 2014.

[40] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of POPL 2011*, pages 357–370. ACM, 2011.

[41] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

[42] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs, 2015. [Online; accessed 11-February-2015].

[43] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL 2003*, pages 185–197. ACM, 2003.

[44] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *Proceedings of ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[45] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proceedings of CSL 2009*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009.

[46] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[47] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, pages 327–336. ACM, 2010.

[48] Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. Available at the journal website, To Appear.

[49] Cosimo Laneve. A type system for JVM threads. *Theoretical Computer Science*, 290(1):741 – 778, 2003.

[50] Sajee Mathew. Overview of amazon web services. at `d0.awsstatic.com/whitepapers/aws-overview.pdf`, November 2014.

[51] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE 2009)*, pages 386–396. ACM, 2009.

[52] Jorge Navas, Mario Méndez-Lojo, and Manuel V Hermenegildo. Customizable resource usage analysis for java bytecode. Submitted, 2008.

[53] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[54] Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the Java Virtual Machine Language: 1. In *Proceedings of the 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, page 10. ACM, 2007.

[55] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[56] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[57] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[58] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[59] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*, pages 602–629. Springer, 2005.