Thesis for the Degree of Doctor of Engineering

# Practical, Flexible Programming with Information Flow Control


## Niklas Broberg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2011

Practical, Flexible Programming with Information Flow Control
Niklas Broberg

© Niklas Broberg, 2011

Technical report 80D
ISSN 0346-718X
Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers, Göteborg, Sweden, 2011

# Practical, Flexible Programming with Information Flow Control

**Niklas Broberg**

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

## Abstract

Mainstream mechanisms for protection of information security are not adequate. Most vulnerabilities today do not arise from deficiencies in network security or encryption mechanisms, but from software that fails to provide adequate protection for the information it handles. Programs are not prevented from revealing too much of their information to actors who can legitimately interact with them, and restricting access to the data is not a viable solution. What is needed is mechanisms that can control not only what information a program has access to, but also how the program handles that information once access is given.

This thesis describes Paralocks, a language for building expressive but statically verifiable fine-grained information flow policies, and Paragon, an extension of Java supporting the enforcement of Paralocks policy specifications. Our contributions can be categorised along three axes:

- The design of a policy specification language, Paralocks, that is expressive enough to model a large number of different mechanisms for information flow control.
- The development of a formal semantic information flow model for Paralocks that can be used to prove properties about programs and enforcement mechanisms.
- The development of Paragon, an extension of Java with support for enforcement of Paralocks information flow policies.

Together these components provide a complete framework for programming with information flow control. It is the first framework to bring together all aspects of information flow control including dynamically changing policies such as declassification, making it both theoretically sound as well as usable for solving practical programming problems.

**Acknowledgements** Many people deserve thanks for helping me get to the point where I am today, for helping me complete this work and my degree.

First and foremost I owe huge thanks to my supervisor, Dave Sands. Dave has not only co-authored much of the work in this thesis – with numerous hours of fruitful discussions, brainstorming, and wild races towards paper deadlines – but has also been a steady source of general support and inspiration. His wealth of crucial knowledge and experience has been invaluable in guiding me towards becoming a proper researcher. Not the least, I am grateful for his willingness to believe in a young student with a grand idea, and for letting me run with it. Besides all this, Dave has also been a great friend and an invaluable support during times of personal hardship. Dave, you have been nothing short of awesome. I am truly grateful for everything you have given me, and I greatly look forward to continuing that work together.

I also owe much to Rogardt Heldal, who has been a great friend and motivator all the way. He has helped me focus on the important, and has inspired me not the least in my aspirations to become a good teacher.

The rest of my PhD committee, Andrei Sabelfeld and John Hughes, have provided many excellent suggestions and feedback on this thesis and previous work.

My opponent Stephan Zdancewic provided useful feedback on my thesis drafts, and will no doubt give me many interesting challenges in defending my work.

Being a graduate student in the department has been a real pleasure, and I have appreciated my time here immensely, even if I have not always been as present as I would have preferred. Many thanks to Josef Svenningsson, who was the one to awaken in me the thrill of doing research. Without his friendship, support and mentoring, I would not have gotten to where I am today. To Ulf Norell and Nils Anders Danielsson, for being great room-mates, and for giving me all the left-overs. To Aslan Askarov, for inadvertently inspiring me to invent flow locks. To Daniel Hedin, for being a steady source of interesting discussions on life, politics and general stupidities. To Phu Phung, for challenging my mediocre badminton skills. And to everyone else in the research group and the department at large, who help create such a friendly and creative atmosphere. I am very glad that my time here will not be over for some time yet.

Huge thanks also to my family who have always believed in me and supported me over the years. To my parents, Anita and John-Olof, for the endless encouragement and support since the beginning of my days, and for providing a loving sanctuary whenever I needed it. To my brother Pontus, the best brother anyone could ever ask for. To my grand-parents May-Britt and Karl-Axel, whose pride in me encourages me to grandeur. To Anne-Lill, who never failed to understand and support my work, even when our roads led us apart.

And last but quite the opposite of least, my beloved Sophia. You caught me, you held me, you weathered the storms with me, and now you'll never be rid of me. Your love and support gives me the wings that make me fly. I love you.

# Table of Contents


**1 Introduction** **11**
- 1.1 Information Flow Control . . . 12
- 1.2 Language-Based Security . . . 15
- 1.3 A History of Information Flow Control . . . 17
- 1.4 Thesis Contributions . . . 20
  - 1.4.1 Thesis Organisation . . . 21
  - 1.4.2 General Contributions . . . 24
  - 1.4.3 Author Contribution . . . 24

**2 Flow Locks** **25**
- 2.1 Introduction . . . 25
- 2.2 Motivating Examples . . . 26
- 2.3 Flow Lock Security . . . 29
  - 2.3.1 Preliminaries . . . 32
  - 2.3.2 Motivating the Security Definition . . . 34
  - 2.3.3 Flow Lock Security . . . 36
- 2.4 Basic Properties of Flow Lock Security . . . 38
- 2.5 Enforcement: A Sound Flow Lock Type System . . . 40
  - 2.5.1 Language . . . 40
  - 2.5.2 Type System . . . 40
- 2.6 Example Encodings . . . 43
  - 2.6.1 Delimited Non-Disclosure . . . 43
  - 2.6.2 Gradual Release . . . 44
  - 2.6.3 More encodings . . . 48

**3 Paralocks** **51**
- 3.1 Introduction . . . 51
- 3.2 Roles and Information Flow . . . 53
- 3.3 Flow Locks and Roles . . . 55
  - 3.3.1 Modeling Roles . . . 56
  - 3.3.2 The Paralocks Policy Language . . . 58
  - 3.3.3 Beyond Roles . . . 60

3.4 Paralocks Security . . . 61
3.4.1 Computation Model . . . 61
3.4.2 Validating flows . . . 62
3.4.3 Paralocks Security . . . 64
3.5 Enforcement: A Sound Paralocks Type System . . . 66
3.5.1 Operational Semantics . . . 69
3.5.2 Type System . . . 72
3.5.3 Security . . . 75
3.6 Example Encodings . . . 76
3.6.1 Robust Declassification . . . 76
3.6.2 The Decentralised Label Model . . . 76
3.7 Recursive Paralocks . . . 81
3.7.1 Policy . . . 81
3.7.2 Expressiveness . . . 82
3.7.3 Semantics . . . 83
3.7.4 Enforcement . . . 84

**4 Paragon** **87**
4.1 Introduction . . . 87
4.1.1 Why Java? . . . 88
4.1.2 Design Guidelines . . . 89
4.2 Example Programs . . . 89
4.2.1 Simple Declassification . . . 89
4.2.2 Robust Declassification . . . 91
4.2.3 Sealed-bid Auctions . . . 93
4.2.4 Lexically Scoped Flows . . . 97
4.3 The Paragon Language . . . 101
4.3.1 Types, Policies and Modifiers . . . 101
4.3.2 Locks . . . 102
4.3.3 Type Parameters . . . 103
4.3.4 Actors and Aliasing . . . 104
4.3.5 Type Methods . . . 105
4.3.6 Exceptions and Indirect Control Flow . . . 105
4.3.7 Field Initialisers . . . 107
4.3.8 Policy Inference and Defaults . . . 109
4.3.9 Runtime Policies . . . 109
4.4 The Paragon Type System . . . 110
4.4.1 Typing Judgment . . . 111
4.4.2 Typing Expressions . . . 116
4.4.3 Typing Statements . . . 121
4.4.4 Typing Blocks and Block Statements . . . 128

4.4.5 Typing Method Declarations . . . 128
4.5 Compiling Paragon . . . 130
4.6 A Comparison with Jif . . . 131
4.6.1 The Jif Language . . . 132
4.6.2 Jif Concerns . . . 133
4.6.3 Feature Comparison . . . 133
4.6.4 Example: Encoding the DLM . . . 135

**5 Related work** **139**
5.1 Policy Specification Mechanisms . . . 139
5.2 Semantics of Information Flow . . . 141
5.3 Information Flow Programming Languages . . . 142
5.4 Typestate Systems . . . 144

**6 Conclusions and Future work** **147**

**A Flow locks: Proofs and auxiliary definitions** **160**
A.1 Type system proofs . . . 160
A.2 DLM encoding . . . 169

**B Paralocks: Proofs and auxiliary definitions** **173**
B.1 Type System Security Proof . . . 173

# Chapter 1

# Introduction

Knowledge has always been power – and today this is more true than ever. Information, and then in particular in digital form, is today bought and sold in enormous quantities on an ever-expanding market, driven not the least by the increasing use of so called social media in our daily lives.

The primary reason for the massive increase in information handling today is of course the ease with which information can be handled in digital form. Digital storage devices the size of your hand could contain the collected information of several libraries' worth of books, and world-wide digital networks makes the spreading of information take only a fraction of the time it once took to copy and deliver the same information on paper.

However, with the increasing importance and abundance of information, there's an obvious equally increasing need for ensuring that it the information is handled correctly. The current trend of computerisation, digitalisation and networking has been accompanied by a dramatic rise in computer security incidents.

The field of *information security* can be described as the art of ensuring that information is handled in a secure fashion, to safeguard the information from incidents. Information security can be divided into three broad aspects:

- Information confidentiality – the task of ensuring that information is only available to those who should have access to it.
- Information integrity – the task of ensuring that information is not manipulated in unintended ways.
- Information availability – the task of ensuring that the information exists where it is supposed to, when it is supposed to.

The last of these, information availability, is typically an issue of *system* availability, where the information itself plays a secondary role. Thus, when we

in the remainder of this introduction refer to the term information security, we mean the first two aspects – confidentiality and integrity.

## 1.1 Information Flow Control

Envision in a non-computerised setting, a company having trade secrets - e.g. prototypes, research documents or the like. Clearly they do not want just anyone to be able to take part of those secrets - there is a need to keep them confidential, a need for information security. *How* the company goes about ensuring the confidentiality of its secrets is a matter of *enforcement*. Likely there will be several measures involved. For starters, the secrets would surely be kept in such a way that only trusted employees could access them. They would be guarded by locked doors, needing keys and codes to get past. There might even be physical guards confirming the identity of anyone coming in.

Also when some secrets for some reason need to *intentionally* leave the facilities, there would likely be procedures for how they should be handled - locked bags, guards and escort cars are all possible measures, depending on the potency of the secrets.

These measures to control access to the secrets would not be enough though. If physical access was the only thing controlled, there would be nothing to stop an employee with access to simply walk out the facilities with some secrets in their bag, either by clumsy mistake or through malicious intent. To prevent against such incidents, the company likely needs to employ measures to control how secrets are handled when accessed. There may be protocols to adhere to while being in the facilities, to avoid unintentional leaks of the secrets. There may be monitoring, through surveillance cameras and through guards screening the bags of people leaving. Cameras may be forbidden in the facilities. Employees accessing the secrets may need to register the purpose and intent of their access in advance. In fact, it is not inconceivable that the company would employ pre-screening of employees before deciding to trust them, to avoid giving access to someone with malicious intent.

What we have described here are various aspects of information security enforcement. When going to a computerised, network-based setting, there are clear analogies to all these measures. Somewhat crudely, enforced mechanisms can be categorised into three broad domains:

- *Encryption* deals with information security *outside* the system environment, to ensure that the information remains confidential and intact until it reaches its intended recipient. This is analogous to the locked bags and guards used to protect the secrets outside the facilities.

- *Access control* deals with information security at the system *borders*. Its purpose is to restrict access to the system and its information to only those users that may interact with it. Many conventional security mechanisms fall into this category, such as firewalls and password protection mechanisms. This is analogous to the locked doors and guards on the facilities, stopping unauthorised people from entering.

- *Information flow control* deals with information security *inside* the system environment, detailing how the information is used once access has been given. This is the domain of ensuring that the system *software* treats the information in the system in the intended way. All the remaining measures described in the example fall into this category. Protocols, restrictions, screens and monitors, intent control and pre-screening - all are different aspects of information flow control.

From our example it should be clear that the company cannot suitably protect their secrets by using only the equivalents of encryption and access control. Similarly, when looking at the computerised settings, the need for information flow control should be clear. Despite this, traditionally most effort in ensuring information security has been devoted to the first two of these domains. While certainly needed, such measures cannot provide a complete solution to the problem of information security. Software today plays an increasingly dominant role in everything from traditional computers and servers to mobile phones to vehicle systems, and the focus of information security enforcement must shift towards software aspects accordingly. The need to focus on the security of software applications is supported by general vulnerability statistics from the US National Institute of Standards and Technology, December 2006. Figure 1.1 illustrates that 92% of all reported vulnerabilities are in software applications – not in networks or encryption modules. While not all software vulnerabilities can be credited to information flow issues – for example, many are likely to be low-level memory safety problems – the need for a focus shift should still be evident.

There is thus an increasing need for better methods to ensure that software handles information correctly, in accordance with the security requirements – the information security *policy* – of that information. Traditional techniques to information flow control are most often either post-hoc attempts to add a layer of information security to existing systems, or ad-hoc principles to adhere to when writing new software. Both approaches lead to flawed and impotent security schemes, to which the numbers in figure 1.1 are a testament.

When looking at the list of example enforcement measures for information flow control in our example, these are quite diverse. However, we can again

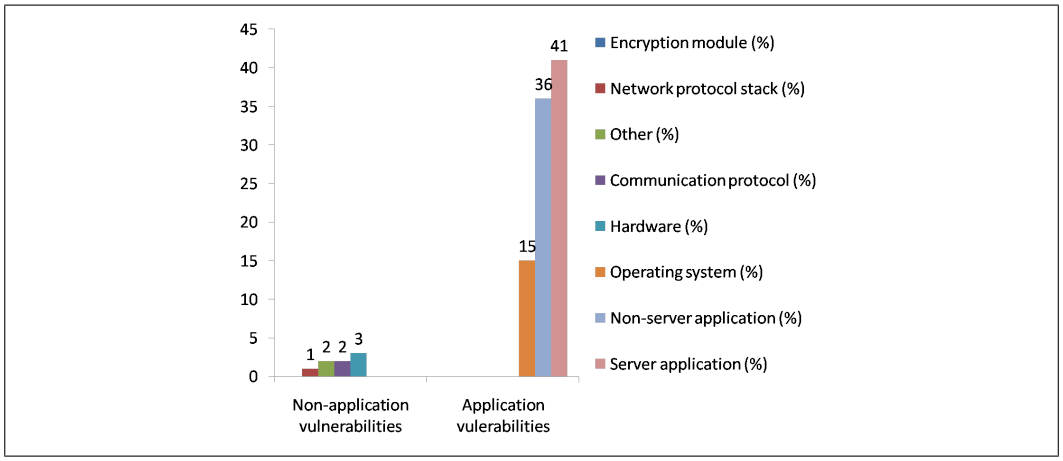



**Figure 1.1:** Vulnerability stats, NIST, 12/2006

categorise these measures along two dimensions: those that control information flow *dynamically* at the time when the information is handled, and those that perform the checks *statically*, by ensuring in advance that the secrets will be handled correctly. The surveillance cameras and screening of bags on exit are clear examples of dynamic monitoring, while the pre-screening and the restriction on cameras are examples of static checks.

When looking at the information flow aspects of software, we can similarly employ both static and dynamic techniques to ensure that information is handled correctly. Dynamic techniques observe how a running program behaves, and would attempt to hinder any potential unallowed uses of information. The analogy with the company example is straight-forward.

For static measures, however, there is a potential in the software setting that has no analogue in our company example[1]. To check a program statically, before it runs, that implies that we have access to the program in some form - source code, byte code or, in the worst case, binary code. That program code is then an *exact* specification of how the program will handle the information it is given access to. Exact may not necessarily mean easy to analyse, but at least in theory we have the potential to perform static analyses of the code, to establish in advance whether or not the code can safely be given access.

[1]It would clearly be inconceivable to perform a similar analysis of a person before granting access, barring significant advances in mind reading technology.

## 1.2 Language-Based Security

Language based security is the domain of analysing and enforcing software information security by employing programming language techniques. The rationale for this approach is that the best way to achieve software security is to ensure that the software is written correctly. By applying analyses and enforcement mechanisms at the programming stage, the goal would be that a program that passes these mechanisms is guaranteed to preserve the security of the information it handles.

Language-based security techniques can be applied to information security aspects in general, but are particularly useful for handling information flow control, i.e. how information flows through a program. There are many different ways in which information can be leaked by a running program. These ways are referred to as information flow *channels*.

When information is leaked directly, by e.g. the program sending the information as is over the network, we refer to this as a *direct* flow channel. Such direct flows could be monitored by dynamic means, without looking at the program code. But information can also be leaked through *indirect* flow channels. Say for example that the program decides which action to perform next based on a secret value. The value may not be transmitted directly as is, but anyone observing the result of the action that was performed, and knowing enough about the program, can deduce something about it anyway.

An analogy could be an employee observing their boss' behavior after an important phone call. If the boss is known to always go for a cup of coffee after receiving bad news, the employee would not need to overhear the phone conversation to know whether or not it was indeed bad news.

Language-based techniques are useful to handle both direct and indirect flows, the latter because having information about *why* a program performed a particular action – e.g. knowing the coffee habits of the boss – requires analysing the source code of the program.

There are many more examples of potential information flow channels, so called *covert* channels. These include observing whether or not a program terminates, observing the time it takes a program to perform a certain task, or measuring its memory consumption. Apart from the termination channel, these covert channels are typically very difficult to secure, and work on information flow control typically ignores such channels since the cost-to-gain ratio is too high. Regardless, language-based techniques may be more or less well-suited for these tasks – but handling *all* potential channels for information leakage requires many different techniques working together anyway. Indeed, all measures for enforcing software security would be worthless against an attacker that physically breaks in to steal the media containing

the secret data. Using language-based techniques to handle some common information flow channels is thus just one piece of a larger security puzzle. But no chain is stronger than its weakest link, and, as indicated by the statistics in figure 1.1, right now that weakest link is in application security, including information flow control.

As noted, language-based security encompasses more than information flow control, and there are information flow channels that are not necessarily best handled by language-based techniques. But for the most common and (arguably) most easily exploitable channels, language-based techniques are particularly well-suited.

Such techniques can be applied at different stages in the process of software creation and execution, depending on what they are trying to ensure or protect. For example, one language-based mechanism could be to preprocess the source code of a program right before it is run by an interpreter, so that it is guaranteed to either not leak during execution (if it was written correctly), or terminate with an error before any leak can take place. Such a measure would come quite late in the process, right before execution, which would have the benefit that it could be applied e.g. in a server environment to programs written by external, untrusted parties. One particular use case for such a measure could be in a browser executing JavaScript code from unknown and/or untrusted sources.

Our focus has been at the other end of the process, at the point where the program is written. We focus on tools to help the programmer verify that their program does not leak unintentionally. To solve that task, the first question that arises is this: When is a program secure? This question must be broken down into three different sub-tasks:

- What is the security *policy* that the program should adhere to?
- By what standard do we judge whether the program adheres to that policy?
- How do we assure that the program lives up to that standard?

These three sub-tasks correspond to the issues of *policy specification*, *semantic security characterisation*, and *policy enforcement*.

In the following section we will look briefly at the history of the notion of information flow policies related to their specification, semantic characterisation and enforcement, to give an overview of the domain to which the work presented in this thesis contributes.

## 1.3 A History of Information Flow Control

The history of information flow control goes back to Bell and LaPadula [BL73], who were the first to create a mathematical model of program security. Their model was based on a chain of *secrecy levels* taken from the military setting – *unclassified* < *classified* < *secret* < *top secret*. These levels were assigned to data, and there were conditions to restrict flows from data at "higher" levels to "lower". While this model is still influential today, as an implicit basis for most enforcement mechanisms of information flow control, they did not provide a semantic characterisation of what it means for a program to be secure.

The history of semantic characterisation of information flow instead goes back to Cohen's work on *strong dependencies* [Coh77, Coh78]. This work is the basis for the notion of *non-interference* that is still today used as the fundamental (total) information flow security requirement. The term "non-interference", however, was coined by Goguen and Meseguer [GM82].

The work that has been the most influential is arguably that by Denning [Den76], who expanded on the model by Bell and LaPadula to allow a *lattice* of levels as the specification of policies. Denning was also the first to explicitly characterise *indirect* flows arising from the control flow of a program. Denning and Denning introduced the notion of *security contexts* to disallow "low" side-effects happening in "high" branches or loops, and the use of a *program counter* to track such contexts in a dataflow analysis [DD77, Den82].

Like Bell and LaPadula, however, Denning's model lacked a semantic characterisation of information security. It would be 20 years until Volpano, Smith and Irvine addressed this problem [VSI96]. They write:

> "So far there has not been a satisfactory treatment of the soundness of Dennings analysis. After all, we want to be assured that if the analysis succeeds for a given program on some inputs, then the program in some sense executes securely. Denning provides intuitive arguments only..."

In their work, Volpano et al presented a semantic non-interference condition for information flow, and proved Denning's enforcement mechanism sound according to

Volpano et al, building on the work by Denning, were thus the first to present all three aspects – specification, characterisation and enforcement – together as part of a coherent model. Denning's lattice model is one of only two models that have been used as a basis for a significant portion of the research on information flow control to date.

**Non-interference and its drawbacks** Non-interference is a semantic condition for information flow security. In simple terms it states that the "high" inputs of a program may not, in any way, influence the program's "low" visible outputs.

The condition is total, i.e. it allows no exceptions. This is a strength in that it allows for precise analyses and enforcement mechanisms to *prove* that a given program satisfies the condition. However, this strength becomes a weakness in practice. Most programs in practice require *some* influence of "high" data on "low" outputs. As a very simple example, consider a basic password checking mechanism. It prompts the user for a (public) guess, compares this guess to its stored (secret) password, and either lets the user in (if the guess was correct) or responds with an error. This very basic program does not fulfill the non-interference condition: the (public) response from the program depends in part on the secret password.

That non-interference is too strict a condition for practical use can be further be argued, anecdotally, from the fate of the language FlowCaml [Sim03], developed by Pottier and Simonet [PS03]. FlowCaml extends the programming language ML with support for information flow control in the form of a Denning-style lattice model. Data is annotated with security levels, and a full information flow type checker, including ML-style *inference* of levels to make programming palatable, guarantees that well-typed programs are secure. The type checker is elegantly proven to indeed guarantee strict non-interference between security levels.

FlowCaml is quite impressive work, yet was practically never adopted for any use, other than as a reference for further research. We surmise, with emphasis, that the reason for this is exactly that non-interference is prohibitively strong as a security requirement in practice. We argue that FlowCaml marks the pinnacle work – and end point – of the original Denning model.

**Declassification** The realisation that non-interference is too restrictive of course does not mean that we must let "high" inputs *arbitrarily* influence "low" outputs. That would mean not caring about information flow control at all. Instead what is needed is a way to specify and enforce policies where programs can *deliberately* let "high" inputs influence "low" outputs in a *controlled* fashion. For instance, for our password checking example it would be fine for the program to reveal whether or not the password matches a given guess, but not fine for the program to reveal the password in full.

The notion of deliberately leaking information is traditionally known as *declassification* – i.e. making data "less classified". The term declassification

implicitly refers to a notion of information flow based on *confidentiality*, owing back to the security levels of the model of Bell and LaPadula. However, information flow also involves issues of *integrity*, which can be argued to be the dual notion of confidentiality. For integrity aspects, the analogy to declassification is called *endorsement*. A more neutral term that includes both declassification and endorsement is *downgrading*. In the remainder of the introduction we will use the term declassification, since it has been most prevalent in the literature we discuss here.

There are many different models of various kinds of declassification, and declassification can be controlled according to several different criteria. Sabelfeld and Sands [SS05] have made a recent study of existing declassification mechanisms in which they categorise mechanisms along four different *dimensions*:

- *What* information is declassified, as in our password example where the whole password may not be leaked.
- *When* information is declassified. Some data may be made available to a user only after they have paid for it.
- *Who* may decide to declassify some information. A patient may decide to share his medical record with his insurance company, but the company should not be able to make that decision.
- *Where* in a program information is declassified. This is a programming notion, where some parts of a program are considered trusted to perform declassifications.

We refer to Sabelfeld and Sands [SS05] for the complete survey.

**The Decentralised Label Model** The model of information flow that has had the most impact on information flow research apart from that by Denning is arguably the *Decentralised Label Model* (DLM), by Myers and Liskov [ML97]. The DLM is a language for specifying information flow policies that allow for controlled declassification along the "who" and "where" dimensions, and is thus inherently less strict than the Denning model. The DLM has been implemented as the policy specification language used in the language *Jif* [MZZ+06]. Jif is an extension of Java that adds information flow control primitives through the inclusion of DLM labels on data, and a type system that statically guarantees information flow properties about programs.

However, the primary weakness of the DLM (and thus Jif) is that, like the old model by Bell and LaPadula, it comes without a semantic characterisation

of security. Since the DLM allows declassification, it is clear that it cannot guarantee non-interference – and in fact we would not want it to, since non-interference is too restrictive. DLM needs a weaker semantic model, one that can account for controlled use of declassification, but no such model exists (prior to this work). This means that while the type system of Jif purports to make some guarantees, we do not know just what those guarantees actually are.

Other policy models incorporating declassification have been proposed, that do include full semantic conditions for (their versions of) information flow security. One example is the work by Almeida Matos and Boudol on *non-disclosure* [AB05], a model which allows localised ("where") declassification using a Denning-style lattice for policy specification. Despite being complete and proven correct, this model, like other similar models, has not become very influential or widely adopted. We surmise that this has three causes:

- Firstly, each model includes only a limited form of declassification, such as the model by Almeida Matos and Boudol only handling the "where" dimension of declassification. While in theory *some* form of declassification is sufficient to allow programs that must deliberately leak information, in practice it may not allow them to be written *conveniently*. Nor is it clear that a particular semantic model is fine-grained enough to be able to represent and guarantee the different dimensions of declassification.

- Secondly, semantic security models for information flow involving declassification tend to be quite complex and unintuitive. The model proposed by Almeida Matos and Boudol is one example of this; our own early attempts were even worse [BS06a]. Compared to the simple and intuitive characterisation of strict non-interference, this is a definite hindrance for the general adoption of any model.

- Lastly, no model has been implemented, like the DLM, as part of a general purpose programming language. The fact that the DLM has been implemented in Jif has allowed it to be used in case studies and courses on computer security, giving hands-on experience. This, we surmise, has allowed the DLM to prevail where other, more fully specified models have not.

## 1.4 Thesis Contributions

What we have described above details the state of the art in which the contributions of this thesis should be put in context. On the one hand we have

the too-strict notion of non-interference, taking off with the work by Denning and, in some sense, culminating with FlowCaml. This line of work has a simple and (relative to its needs) flexible policy specification language (a lattice of security levels); a formal, complete and intuitively simple semantic characterisation of information flow security (non-interference); well-studied and formally proven enforcement mechanisms; and a full-fledged implementation in FlowCaml.

On the other hand we have a diverse plethora of work involving some notion of declassification, to make information flow control practically useful. Some of these mechanisms have formal semantic models of information flow security. Some have clever type systems to enforce security in the presence of declassification. Few focus on policy specification issues, and only one – Jif/DLM – has a full-fledged implementation. None of them manages to combine all these aspects, most only one or two.

The work presented in this thesis incorporates all these aspects, forming a complete platform for information flow security in the presence of declassification. This can thus be stated as the main contribution of this work: It is the first platform for information flow control including declassification that brings all the necessary bits together.

It is important to note that this thesis does not represent an end point, but rather a status report of an ongoing project. Much still remains to be done, even if we have come far enough to refer to our work as a "complete" platform.

### 1.4.1 Thesis Organisation

The thesis is organised into six chapters, of which chapters 2, 3 and 4 hold the main technical results of our work.

Our work up to now has been presented previously in a sequence of four papers, each of which adds a piece of the overall picture. In the presentation below we discuss how each of those papers contribute to this thesis.

**Chapter 1 – Introduction** This chapter, in which we set the context for our work.

**Chapter 2 – Flow Locks** Here we introduce *flow locks*, a policy specification mechanism for dynamic information flow. The chapter is based on two earlier papers. The first is "Flow locks – Towards a Core Calculus for Dynamic Flow Policies" [BS06b], in which we introduce flow locks, and show how they can be used to encode a number of other proposed mechanisms for

declassification, arguing its potential as a stepping stone for a core calculus of policy specification.

In this paper we gave a full semantic model for information flow security related to flow lock policies. Further we showed a type system for a small ML-like language that incorporates flow locks, and proved that the type system guarantees our semantic security condition.

The semantic model given in this paper was the first model to allow dynamic changes to the security policy at arbitrary points in the program, and to allow the policy to become both more liberal or more restrictive. Earlier (and subsequent) models only allow a successively more liberal policy, or policies becoming more liberal in a localised piece of the program.

Both the semantic model and the presented type system were influenced by the work by Almeida Matos and Boudol on non-disclosure, as well as earlier work by Mantel and Sands on *intransitive non-interference* [MS04]. The resulting semantic model was complex, unintuitive and very cumbersome to work with.

The second paper is "Flow-sensitive Semantics for Dynamic Information Flow Policies" [BS09], in which we completely rework the semantic model for flow locks. We base our new model on a *knowledge-based* style inspired by the work on *Gradual Release* by Askarov and Sabelfeld [AS07]. We show how their model for simple two-level policies can be generalised to provide a model for flow locks, including policies that become more restrictive as execution progresses.

We also present a type system for a simple while-language, and prove that it guarantees our new semantic security condition. The semantic model of this paper, as well as the type system given, is presented along with the flow locks specification language in chapter 2.

In this thesis we present the policy specification mechanism as defined in the first paper, but then go on to introduce the semantic model and type system from the second paper.

**Chapter 3 – Paralocks** While we could show that flow locks was flexible enough to encode a number of other mechanisms for information flow control, this was only true in theory. In practice there were issues that made flow locks too inflexible, in particular the requirement that all actors interacting with a program were statically known and enumerable at compile time.

In our third paper, "Paralocks – Role-based Information Flow Control and Beyond" [BS10], we extend the flow locks mechanism to solve these shortcomings. We show that the extended language, dubbed *Paralocks* (*para*meterised *locks*), can encode other mechanisms in a practical way, and thus does not

suffer from the drawbacks of its predecessor.

We also extend the semantic security model from the previous paper accordingly, to allow for actors not known until run-time. Like previously we also show a type system for a simple while-language, and prove that it guarantees that well typed programs are secure.

An interesting detail is that one of the mechanisms we show an encoding of is the DLM. Thus we are able to give the first full semantic characterisation of the DLM, through our encoding into Paralocks.

In this thesis we present the Paralocks policy specification language and its accompanying semantic model, along with the type system and the encodings of other mechanisms.

**Chapter 4 – Paragon** In this chapter we describe how to incorporate Paralocks in a full-fledged general purpose programming language (Java). We call the resulting language *Paragon*. This chapter serves as an extended version of our paper "Paragon for Practical Flow-Oriented Programming" [BS11].

We discuss the issues that arise with features like exceptions and the class hierarchy, and how they affect the typing of Paragon programs. We sketch the implementation of Paragon, including the sketch of a type system that incorporates the most important aspects of checking Paralocks policies in this setting. The main difference from the paper is the presentation of the type system, which was omitted from the paper due to space constraints.

What we do not yet have is a formal proof that our type system for Paragon guarantees the semantic security condition presented in chapter 3.

**Chapter 5 – Related Work** In this chapter we look at related work along the three axes we have pointed out: Policy specification mechanisms, semantics of information flow, and programming languages with information flow control capabilities.

Further, we also look at work on the concept of *typestate* and how it relates to the use of locks in Paragon.

**Chapter 6 – Conclusions and Future Work** In the final chapter we give some concluding remarks on our work, and point out several directions for future research to further improve the platform.

### 1.4.2 General Contributions

We can state our contribution along the three different aspects of information flow control we identified earlier: Specification, semantic characterisation, and practical enforcement:

**Policy specification mechanisms:** We have shown that Paralocks is a simple yet flexible and expressive language for specifying information flow policies in the presence of dynamic changes and declassification. Paralocks can encode a large number of other proposed mechanisms along the "who", "where" and "when" dimensions of declassification – notably including the DLM – thus serving as a core calculus for specifying such policies.

**Semantic models for information flow:** We present a simple and intuitive condition for when programs satisfy the information flow security requirements as specified by a Paralocks policy. A notable contribution is the combination of this point with the previous: Our semantic model relates to an expressive language for information flow policy specification, not just a two-level "high-low" system or a simple powerset lattice of actors.

**Programming with information flow control:** Paragon is only the third full programming language with support for enforcement of information flow control, after Jif and FlowCaml. That in itself is a contribution, but Paragon also improves over these two language in several aspects. Specifically, Paragon allows flexible use of controlled declassification, based on a formal semantic model.

### 1.4.3 Author Contribution

I, Niklas Broberg, the author of this thesis, have been instrumental in the conception, design and development of all the work discussed herein. While all the work is to varying extent joint work with my supervisor David Sands, I have largely been the driving force behind it. The original idea for flow locks was mine, thought up in response to a challenge by Dave to improve the state-of-the-art of information flow control languages. Along the way I have been the main contributor to the semantic models and type systems, and I have been the one doing all the requisite proofs. I had great help from, and many long and fruitful brain-storming discussions with, Dave, and have been expertly guided along by his great wisdom and experience. Yet I can proudly proclaim the work presented in this thesis as primarily mine.

# Chapter 2

# Flow Locks

## 2.1 Introduction

Unlike access control policies, enforcing an information flow policy at run time is difficult because information flow is not a runtime property; we cannot in general characterise when an information leak is about to take place by simply observing the actions of a running system. From this perspective, statically determining the information-flow properties of a program is an appealing approach to ensuring secure information flow. However, security *policies*, in practice, are rarely static: a piece of data might only be untrusted until its signature has been verified; an activation key might be secret only until it has been paid for. In more formal terms, there is a need for *downgrading* of information.

In this chapter we introduce a simple policy specification mechanism based on the idea that the reading of variable $x$ by certain actors (principals, levels) is guarded by boolean flags, which we call *flow locks*. For example, the policy $x_{\{high; Paid \Rightarrow low\}}$ says that $x$ can always be read by an actor with a high clearance level, and also by an actor with a low clearance level providing the "Paid" lock is open.

The interface between the flow lock policies and the security relevant parts of the program is provided by simple instructions for opening and closing locks. The program itself does not depend on the lock state, and the intention is that by statically verifying that the dynamic flow policy will not be violated, the lock state does not need to be computed at run time.[1]

In addition to the introduction of the Flow locks policy specification lan-

[1] The term *dynamic* flow policy could have different interpretations. We use it in the sense that the flow policies vary over time, but they are still statically known at compile time.

guage, we will also discuss a number of its features:

- A formulation of the semantics of secure information flow for flow locks.
- The definition of a type system for a simple while language which permits the completely static verification of flow lock policies, and a proof that well typed programs are flow-lock secure.
- The demonstration that flow lock policies can represent a number of other proposed information flow paradigms.

Regarding the last point, the work presented here can be viewed as a study of *declassification* mechanisms. In a recent study by Sabelfeld and Sands [SS05], declassification mechanisms are classified along four dimensions: *what* information is released, *who* releases information, *where* in the system information is released, and *when* information can be released. One of the key challenges stated in that work is to *combine* these dimensions. In fact, combination is perhaps not difficult; the real challenge is to combine these dimensions without simply amassing the combined complexities of the contributing approaches. Later in this chapter we argue that flow locks can encode a number of other proposed "declassification" paradigms, including Barthe's et al *delimited non-disclosure* [BCR08], Chong and Myers' notion of *noninterference until declassification* [CM04], and Zdancewic and Myers *robust declassification* [ZM01, MSZ04]. These examples, represent the "where", "when" and "who" dimensions of declassification, respectively, suggesting that flow locks have the potential to provide a core calculus of dynamic information flow policies.

## 2.2 Motivating Examples

```
int aBid = getABid();
int bBid = getBBid();
makePublic(aBid);
makePublic(bBid);
// ... decide winner + sell item
```

First let us assume we have a simple imperative language without any security control mechanisms of any kind. Borrowing an example from Chong and Myers [CM04], suppose we want to implement a system for online auctions with hidden bids in this language. We could write part of this system as the code on the right.

This surely works, but there is nothing in the language that prevents us from committing a serious security error. We could for instance accidentally switch the lines 2 and 3, resulting in $A$'s bid being made public before $B$ places her bid, giving $B$ the chance to tailor her bid after $A$'s.

Flow locks are a mechanism to ensure that these and other kinds of programming errors are caught and reported in a static check of the code.

The basic idea is very similar to what many other systems offer. To deny the flow of data to places where it was not meant to go, we annotate variables with policies that govern how the data held by those variables may be used. Looking back on our example, a proper policy annotation on the variable `aBid` could be $\{A; \texttt{BBid} \Rightarrow B\}$. The intuitive interpretation of this policy is that the data held by variable `aBid` may always be accessed by $A$, and may also be accessed by $B$ whenever the condition `BBid`, that $B$ has placed a bid, is fulfilled. `BBid` here is a *flow lock* — only if the lock is *open* can the data held by this variable flow to $B$. To know whether the lock is open or not we must look at how the methods for getting the bids could be implemented.

```
getABid(){
  int {A; BBid => B} x
    = bidChanFromA;
  open ABid;
  return x;
}
```

The method shown on the right first fetches the bid sent by $A$. We model the incoming channel as a global variable that can be read from, one with the same policy as `aBid`. When the bid has been read, the method signals this by opening the `ABid` lock—$A$ has now placed a bid and the program can act accordingly. The implementation of `getBBid` follows the same pattern, and will result in `BBid` being open. Now both bids have been placed and can thus be released.

The `makePublic` method would be implemented as follows:

```
makePublic(bid){
  publicChannel = bid;
}
```

The outgoing `publicChannel` is also modeled as a global variable that can be written to. This one has the policy $\{A; B\}$ attached to it, denoting that both $A$ and $B$ will be able to access any data written into it. At the points in the program where `makePublic` is applied, both $A$ and $B$ will have placed their bids, the locks `ABid` and `BBid` will both be open, and the flows to the public channel will both be allowed. However, if the lines 2 and 3 were now

accidentally switched, it would be a different story. Then we would attempt to release $A$'s bid, guarded by the policy $\{A; \texttt{BBid} \Rightarrow B\}$, onto the public channel with policy $\{A; B\}$. Since the flow lock `BBid` will then not yet be opened, this flow is illegal and the program can be rejected.

```
auctionItem(firstItem);
aBid = getABid();
bBid = getBBid();
makePublic(aBid);
makePublic(bBid);
// ... decide winner + sell item
auctionItem(secondItem);
aBid = getABid();
bBid = getBBid();
makePublic(aBid);
makePublic(bBid);
// ... decide winner + sell item
```

Taking the example one step further, assume that we have two items up for auction, one after the other. We can implement this rather naively as the program to the right. The locks `ABid` and `BBid` will both be opened on the first calls to the `getXBid` methods. But unless we have some means to reset them, there is again nothing to stop us from accidentally switching lines to make our program insecure, this time lines 9 and 10. The same problem could also be seen from a different angle: what if the locks were already open when we got to this part of the program? Clearly we need a closing mechanism to go with the open. The method `auctionItem` could then be implemented as shown here.

```
auctionItem(item){
  close ABid, BBid;
  // ... present item ...
}
```

By closing the locks when an auction is initiated, we can rest assured that both $A$ and $B$ must place new bids for the new item before either bid is made public.

It should be fairly easy to see that what we have here is a kind of state machine. The state at any program point is the set of locks that are open at that point, and the open and close statements form the state transitions. A clause $\sigma \Rightarrow A$ in a policy means that $A$ may access any data guarded by that policy in any state where $\sigma$ is open.

Our lock-based policies also give us an easy way to separate truly secret

data from data that is currently secret, but that may be released to other actors under certain circumstances. Assume for instance that payment for auctioned items is done by credit card, and that the server stores credit card numbers in memory locations `aCCNum` and `bCCNum` respectively. Assume further that the line `aBid := aCCnum;` is inserted, either by sheer mistake or through malicious injection, just before where `aBid` is made public. This would release $A$'s credit card number to $B$, however, the natural policy on `aCCNum` would be $\{A\}$, meaning only $A$ may view this data, ever. Thus when we attempt the assignment above, it will be statically rejected since the policy on `aBid` is too permissive.

All the above are examples of policies to track confidentiality. The dual of confidentiality is integrity, i.e. deciding to what extent data can be trusted, and it should come as no surprise that flow locks can handle both kinds.

Returning to the example with the credit card, we assume that when $A$ gives her credit card number, it must be validated (in some unspecified way) before we can trust it. To this end we introduce a "pseudo" actor $T$ (for "trusted") who should only be allowed to read data that is fully trusted. We then use an intermediate location `tmpACCNum` to hold the credit card number when it is submitted by $A$. This location is given the policy $\{A; \texttt{ACCVal} \Rightarrow T\}$, stating that this data is trusted only if the lock `ACCVal` is open, which is done when the submitted number has been validated. Once validated we can transfer the value to `aCCNum`, which now has the policy $\{A; T\}$ stating that this data is trusted.[2]

## 2.3 Flow Lock Security

Information flow policies are only useful if we have a precise specification – a semantic model – of what we are trying to enforce. A semantic model gives us insight into what a policy actually guarantees, and defines the precise goals of any enforcement mechanism.

Unfortunately, semantic models of declassification – in particular those that try to specify more that just *what* is declassified – can be both inaccurate and difficult to understand.

**The Flow Sensitivity Problem** The most commonly used semantic definition of secure information flow – at least in the language-based setting – involves the comparison of two runs of a system. The idea is to define security by comparing any two runs of a system in environments that only

[2] In order to prevent overwriting this data with a new number that hasn't been validated, we should also be sure to close the lock `ACCVal` once the assignment is done.

differ in their secrets (such environments are usually referred to as being *low equivalent*). A system is secure or *non-interfering* if any two such runs are indistinguishable to an attacker. These "two run" formulations relate to the classical notion of *unwinding* in [GM82].

Many semantic models for declassification – in particular those which have a "where" or "when" dimension [SS05] – are built from adaptations of such a two-run noninterference condition. [3]

Such adaptations are problematic. Consider the first point in a run at which a declassification occurs. From this point onwards, two runs may very well produce different observable outputs. A declassification semantics must constrain the difference at the declassification point in some way (this is specific to the particular flavour of declassification at hand), and further impose some constraint on the remainder of the computation. So what constraint should be placed on the remainder of the computation? The prevailing approach to give meaning to declassification (e.g. [MS04, EP05, EP03, AB05, Dam06, MR07, BCR08, LM08]) is to reset the environments of the systems so as to restore the low-equivalence of environments at the point after a declassification.

We refer to this as the *resetting approach* to declassification semantics.

The down-side of the resetting approach is that it is *flow insensitive*. This implies that the security of a program $P$ containing a reachable subprogram $Q$ requires that $Q$ be secure independently of $P$. For example, consider the program

$$\text{declassify } h \text{ in } \{\ell := h\}; \ell := h$$

where $h$ is a high security variable and $\ell$ is low. In the semantics of e.g. Barthe et al [BCR08] this would be deemed insecure because of the insecure subprogram $\ell := h$ – even though in all runs this subprogram will behave equivalently to the obviously secure program $\ell := \ell$. Similar examples can be constructed for all of the approaches cited above. Another instance of the problem is that dead code can be viewed as semantically significant, so that a program will be rejected because of some insecure dead code. Note that flow insensitivity might be a perfectly reasonable property for a particular *enforcement* mechanism such as a type system – but in a sequential setting it has no place as a fundamental semantic requirement.

The resetting approach is not without merits though. In particular it is able to handle shared-variable concurrency in a compositional way [MS04, AB05]. However, the use of resetting for compositionality and its use for giv-

[3] For the purposes of this paper it is useful to view declassification as a particular instance of a dynamic information flow policy in which the information flow policy becomes increasingly liberal as computation proceeds.

ing a semantics to declassification are orthogonal, and the flow insensitivity problem carries over to those parts of the environment which are not shared across threads.

The first semantic model we used for flow locks [BS06b, BS06a] suffered from the flow insensitivity problem described above. Perhaps due to its generality it was also overly complex and unintuitive. The key to recovering flow sensitivity and to drastically simplifying the semantics has been to follow the lead of Askarov and Sabelfeld [AS07] who move away from a "two run" view of security semantics, and focus instead on how an explicit representation of the attacker's knowledge evolves as computation proceeds.

**A Knowledge-based Approach** One of the fundamental difficulties in the bisimulation-style definition is that it builds on a comparison between two runs of a system. While this is fairly intuitive for standard noninterference, in the presence of policy changes such as the opening or closing of locks (in our work) or declassification (in other work) it can be hard to see how the semantic definition really relates to what we can say about an attacker.

A more recent alternative to defining the meaning of declassification is to use a more explicit attacker model whereby one reasons about what an attacker learns about the initial inputs to a system as computation progresses [AS07]. The formulation we use here will be closest to that presented by Askarov et al [AHSS08].

The basic idea builds on a notion of noninterference described by Dima et al [DEG06] and can be explained when considering the simple case of noninterference between an initial memory state, which is considered secret, and public outputs. The model assumes that the attacker knows the program itself $P$. Now suppose that the attacker has observed some (possibly empty) trace of public outputs $t$. In such a case the attacker can, at best, deduce that the possible initial state is one of the following:

$$K_1 = \{N \mid \text{ Running } P \text{ on } N \text{ can yield trace } t \}$$

Now suppose that after observing $t$ the attacker observes the further output $u$. Then the attacker knowledge is

$$K_2 = \{N \mid \text{ Running } P \text{ on } N \text{ can yield trace } t \text{ followed by } u \}$$

We will call $K_1$ and $K_2$ *knowledge sets*, and order knowledge sets by $K \sqsubseteq K' \iff K' \subseteq K$. Note that in the above $K_1 \sqsubseteq K_2$: the attackers knowledge increases as the computation proceeds. However, for the program to be considered non-interfering, in all such cases we must have $K_1 = K_2$, since

we require the knowledge to not increase at all throughout the program execution.

This style of definition is the key to our new flow lock semantics. The core idea will be to determine what part of the knowledge must remain constant on observing the output $u$ by viewing the trace from the perspective of the lock-state in effect at that time.

In this section we motivate our flow sensitive definition of flow-lock security. The definition is phrased in terms of a labeled transition system where labels represent observable events. We assume an imperative computation model involving commands and stores (memories), but the definition is otherwise not specific to a particular programming language.

### 2.3.1 Preliminaries

We begin by recalling the precise language of policies and introduce the base assumptions about the operational semantics of the language.

**Policies** In general a *policy* $p$ is a set of *clauses*, where each clause of the form $\Sigma \Rightarrow \alpha$ states the circumstances ($\Sigma$) under which actor $\alpha$ may view the data governed by this policy. $\Sigma$ is a set of locks which we name the *guard* of the clause, and interpret it as a conjunction. Thus for the guard to be satisfied, all the locks $\sigma \in \Sigma$ must be open.

In concrete examples we will often simplify the notation, so that for example we will write (as we did in the introduction to this chapter)

$$\{vendor; Paid \Rightarrow customer\}$$

instead of

$$\{\emptyset \Rightarrow vendor; \{Paid\} \Rightarrow customer\}.$$

A policy $p$ is *less restrictive* than a policy $q$, written $p \sqsubseteq q$, if for every clause $\Sigma \Rightarrow \alpha$ in $q$ there is a clause $\Sigma' \Rightarrow \alpha$ in $p$ where $\Sigma' \subseteq \Sigma$. For example, $\{vendor; customer\}$ is less restrictive than $\{vendor; Paid \Rightarrow customer\}$ which in turn is less restrictive than $\{vendor\}$. We use the distinguished value $\bot$ to denote the least restrictive policy, for variables that all actors can see at all times. The opposite is the policy $\top$, which is simply the empty set of clauses, meaning no actor could ever see the data of a variable marked with that policy. To join two policies means combining their respective clauses. We define

$$p_1 \sqcup p_2 \equiv \{\Sigma_1 \cup \Sigma_2 \Rightarrow \alpha \mid \Sigma_1 \Rightarrow \alpha \in p_1,\ \Sigma_2 \Rightarrow \alpha \in p_2\}$$

It should be intuitively clear that the join of two policies is at least as restrictive as each of the two operands, i.e. $p \sqsubseteq p \sqcup p'$ for all $p$, $p'$. In contrast, forming the union of two policies, i.e. the meet, corresponding to $\sqcap$, makes the result less restrictive, so we have $p \sqcap p' \sqsubseteq p$ for all $p$, $p'$. Both $\sqcap$ and $\sqcup$ are clearly commutative and associative.

We also need the concept of a policy specialised (normalised) to a particular lock state, denoted $p(\Sigma)$, meaning the policy that remains if we remove from all guards the locks which are present in $\Sigma$. So for example, if $p$ is $\{Paid \Rightarrow customer\}$, then $p(\{Paid\}) = \{customer\}$. Formally, $p(\Sigma) = \{(\Delta \backslash \Sigma) \Rightarrow \alpha \mid \Delta \Rightarrow \alpha \in p\}$.

**Operational Semantics** To keep our presentation reasonably concrete we will consider imperative computation modeled by a standard small-step operational semantics defined over configurations of the form $\langle \Sigma, c, M \rangle$ where $c$ ($c'$, $d$ etc.) is a command, $M$ is a memory (store) – a finite mapping from variables to values, and $\Sigma$ is the lock state – the set of locks that are currently open.

We assume that each channel and variable $x$, $y$, ... is assigned a fixed policy, where $pol(x)$ denotes the policy of $x$.

Transitions in the semantics are labeled $\langle \Sigma, c, M \rangle \xrightarrow{\ell} \langle \Delta, d, N \rangle$ where $\ell$ is either a distinguished *silent* action $\tau$, or an *observable* action of the form $x(v)$, where $x$ is a channel and $v$ is the value observed on that channel. We let $w$, $w'$ etc range over observable actions, and $\vec{w}$ a vector of such. We assume the existence of commands which change the lock state. The open and close commands used in section 2.2 are sufficient, although other lock-state changing commands are possible. We do, however, assume that whenever the lock state changes then there is no output or memory change, i.e. if $\langle \Sigma, c, M \rangle \xrightarrow{\ell} \langle \Delta, d, N \rangle$ $\Sigma \neq \Delta$ then we must have $M = N$ and $\ell = \tau$. Given the labeled transition system we define some auxiliary notions.

**Definition 1** (Visibility)**.**

- We say that $x$ *may be visible to* $\alpha$ if $\Sigma \Rightarrow \alpha \in pol(x)$ for some $\Sigma$; otherwise we say that it is *never visible.*

- We say that $x$ *is visible to* $\alpha$ *at* $\Delta$ if $\Sigma \Rightarrow \alpha \in pol(x)$ for some $\Sigma \subseteq \Delta$; otherwise we say that it is *not visible at* $\Delta$.

We extend these definitions to outputs $x(v)$ in the same way, and we say that the silent output $\tau$ is never visible.

### 2.3.2 Motivating the Security Definition

To motivate our definition we will first look at some properties that we expect it to have. First we consider the case of simple declassification from the introduction. Consider the program $\ell := \mathit{declassify}(h); \ell := h$, which would be encoded as as

$$\textbf{open}\ \mathit{Decl}; \ell := h; \textbf{close}\ \mathit{Decl}; \ell := h$$

The intended meaning of closing a lock is *not* that an actor should forget all they learned while the lock was open. Thus we expect this program to be considered secure, since the value of $h$ is already known at the point of the second assignment. In other words, we expect our definition to be flow *sensitive*, as opposed to our old, bisimulation-based definition. Practically this means that our semantic definition cannot be a purely local stepwise definition, but requires us to inspect all knowledge gained by an attacker up to a certain assignment. Then we must validate that assignment in the context of the attacker having that knowledge.

Another feature to note is that our flow locks system allows fine-grained flows, in which a secret may be leaked in a series of unrelated steps. The following policy and program exhibits this:

$$x : \{\{\mathit{Day}, \mathit{Night}\} \Rightarrow \alpha\} \quad y : \{\mathit{Night} \Rightarrow \alpha\} \quad z : \{\alpha\}$$
$$\textbf{open}\ \mathit{Day};\ y := x\ ; \textbf{close}\ \mathit{Day};\ \textbf{open}\ \mathit{Night};\ z := y$$

Here (and in subsequent examples) we assume each assignment generates an observable action – i.e. each variable is viewed as an output channel. Here the secret contained in $x$ is leaked into $z$ via $y$. But at the point where the assignment to $z$ is made, the lockstate in effect does not allow a direct flow from $x$ to $z$ since $\mathit{Day}$ is closed. In addition, at the point where the assignment to $y$ is made, $y$ is not visible at the current lockstate.

To ensure correct information flow in a program, all flows must be validated at each possible "level" that data can flow to. This is not specific to our setting, but a very general statement regarding information flow control. Each of these levels can be thought of as a potential attacker. For each such attacker, we must ensure that the attacker does not learn more than intended about the initial data.

The way to do this is, for each possible attacker, to split the state into a *high* and a *low* portion – the low portion being the part directly visible to the attacker. The security goal is to ensure that the attacker, by observing the low part, does not learn more than intended about the *high* part of the state. For standard noninterference the goal is that the attacker learns nothing. For

gradual release [AS07] the goal is to ensure that nothing is learned for the observations that are not labeled as declassifications.

For our setting, a "level" that we must validate the flows at corresponds to a certain set of locks guarding a location from a particular actor. We note that these levels correspond to the points in the lattice $Actors \times \mathcal{P}(Locks)$.

This leads us to our formal attacker model:

**Definition 2** (Attacker)**.** An attacker $A$ is a pair of an actor $\alpha$ and a set of locks $\Delta$, formally

$$A = (\alpha, \Delta) \in Actors \times \mathcal{P}(Locks)$$

We refer to the lockstate component of an attacker as his *capability*, and assume that $A$ can observe locations guarded from $\alpha$ only by locks in $\Delta$.

Intuitively we may think of an attacker as an actor who may open the locks $\Delta$ at some point in the future, leading to a *future-sensitive* model that enables us to build secure commands by sequential composition from secure commands (see Section 2.4).

We define attacker visibility as a natural extension of actor visibility, by saying that $x$ is visible to $A = (\alpha, \Delta)$ iff $x$ is visible to $\alpha$ at $\Delta$.

For each attacker we then define the *$A$-observable transition* $\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle$ by absorbing transitions which are not visible to attacker $A$.

**Definition 3** ($A$-observable transitions)**.** We can define the transition relation $\xrightarrow{w}_A$ as the least relation satisfying the following rules:

$$\frac{\langle \Sigma, c, M\rangle \xrightarrow{w} \langle \Delta, d, N\rangle \qquad w \text{ is visible to } A}{\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}$$

$$\frac{\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Sigma', c', M'\rangle \qquad \ell \text{ is not visible to } A \quad \langle \Sigma', c', M'\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}{\langle \Sigma, c, M\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}$$

Combining $A$-transitions gives us a useful compound which we denote a *trace*:

**Definition 4** ($A$-observable trace)**.** We define $\langle \Sigma, c, M\rangle \Longrightarrow_A \langle \Delta, d, N\rangle$ if there is a sequence of zero or more transitions from $\langle \Sigma, c, M\rangle$ to $\langle \Delta, d, N\rangle$ with labels not visible to $A$. Now we define the $A$-observable trace $\xRightarrow{\vec{w}}_A$ for some sequence of output labels $\vec{u}$ by equating $\xRightarrow{\varepsilon}_A$ with $\Longrightarrow_A$ (where $\varepsilon$ denotes the empty vector), and by inductively defining

$$\frac{\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}}_A \langle \Sigma', c', M'\rangle \xrightarrow{w}_A \langle \Delta, d, N\rangle}{\langle \Sigma, c, M\rangle \xRightarrow{\vec{w}w}_A \langle \Delta, d, N\rangle}$$

We use the notation $\langle \Sigma, c, M \rangle \stackrel{\vec{w}}{\Longrightarrow}_A$ as a shorthand for $\exists \Delta, d, N.\, \langle \Sigma, c, M \rangle \stackrel{\vec{w}}{\Longrightarrow}_A \langle \Delta, d, N \rangle$, i.e. when we don't care what the resulting configuration is.

To reason about attacker knowledge we need to be able to focus on the parts of a memory which are visible to a given attacker.

**Definition 5** ($A$-low memory, $A$-equivalence)**.**
Memory $L$ is $A$-low for some attacker $A$ if $dom(L) = \{x \mid x \text{ is visible to } A\}$. We say that two memories $M$ and $N$ are $A$-equivalent, written $M \sim_A N$ if their $A$-low projections are identical – i.e. they agree on all variables that $A$ can see.

We will adopt the convention that $M$ and $N$ will range over total memories (i.e. their domain will be the set of all variables). With this we can formalise the notion of attacker *knowledge* as follows:

**Definition 6** (Attacker knowledge)**.**
The knowledge gained by an attacker $A = (\alpha, \Delta)$ from observing a sequence of outputs $\vec{u}$ of a program $c$ starting with a $A$-low memory $L$ written $k_A(\vec{u}, c, L)$, is defined to be the set of all possible starting memories that could have lead to that observation:

$$k_A(\vec{u}, c, L) = \{M | M \sim_A L, \langle \Sigma, c, M \rangle \stackrel{\vec{w}}{\Longrightarrow}_A\}$$

### 2.3.3 Flow Lock Security

With this attacker model in hand, we can now formalise our security requirement. Intuitively, for a program to be flow lock secure we must consider the perspective of each possible attacker $A$, and how his knowledge of the initial memory evolves as he observes successive outputs.

The requirement for each output thus observed is that knowledge of the initial memory only increases if the attacker's inherent capabilities are weaker than the program lockstate in effect at the time of the output. The intuition here is that an attacker whose capability includes the program lock state in effect should already be able to see the locations used when computing the value that is output. Thus no knowledge should be gained by such an attacker. To formalise this intuition we first, for convenience, introduce the notion of a *run*. A run is just an output trace together with the lockstate in effect at the time of the last output in the sequence.

**Definition 7** ($A$-observable runs)**.** The set of all runs of a command $c$ starting with lock state $\Sigma$ and with a starting memory whose $A$-low projection is

$L$, are defined

$$Run_A(\Sigma, c, L) = \{(\vec{u}u, \Delta) \mid M \sim_A L, \\ \langle \Sigma, c, M\rangle \stackrel{\vec{w}}{\Longrightarrow}_A \langle \Sigma', c', M'\rangle \stackrel{w}{\rightarrow}_A \langle \Delta, d, N\rangle\}$$

We can now define our security requirement in terms of runs as follows:

**Definition 8** ($\Sigma$ Flow Lock Security)**.** A program $c$ is said to be $\Sigma$-flow lock secure, written $\mathbf{FLS}(\Sigma, c)$, iff for all attackers $A = (\alpha, \Delta)$, all $A$-low memories $L$, and all runs $(\vec{u}u, \Omega) \in Run_A(\Sigma, c, L)$ such that $\Omega \subseteq \Delta$ we have

$$k_A(\vec{u}u, c, L) = k_A(\vec{u}, c, L)$$

This definition directly captures the intuition that we started out with. An attacker whose capabilities includes the current lockstate in effect at the time of the output should learn nothing new when observing that output. Attackers who do not fulfill this criterion have no constraint on what they may learn at this step. But note that this cannot lead to unchecked flows because we quantify over *all* attackers including, in particular, those with sufficient capabilities.

At the top level we can define security for a self-contained program, i.e. one that doesn't assume any locks are open before it starts:

**Definition 9** (Top-level Flow Lock Security)**.** A program $c$ is said to be flow lock secure, written $\mathbf{FLS}(c)$, iff the program is $\emptyset$-flow lock secure, i.e. $\mathbf{FLS}(\emptyset, c)$.

The above definitions are termination *sensitive*, since they require that no knowledge is gained by the simple observation that there is an output at all. Following Askarov et al [AHSS08] we can define a termination *insensitive* version:

**Definition 10** (Termination Insensitive Flow Lock Security)**.** A program $c$ is said to be termination-insensitive $\Sigma$-flow lock secure, written $\mathbf{FLS_{TI}}(\Sigma, c)$ iff for all attackers $A = (\alpha, \Delta)$, all $A$-low memories $L$, and any two runs $(\vec{u}u, \Omega)$ and $(\vec{u}u', \Omega')$ in $Run_A(\Sigma, c, L)$ such that $\Omega \subseteq \Delta$ we have that

$$k_A(\vec{u}u, c, L) = k_A(\vec{u}u', c, L)$$

In this variant we allow some knowledge to be gained by the last step of the output, but no more than simply learning that there *is* an observable output. See Askarov et al [AHSS08] for more details. Note that by symmetry we compare the knowledge sets under both $\Omega$ and $\Omega'$.

## 2.4 Basic Properties of Flow Lock Security

In this section we look at some basic properties of the definition of flow lock security. We inspect the basic properties of the definition via the *principles of declassification* as stated by Sabelfeld and Sands [SS05], since flow locks are intended to model various forms of declassification (or more generally *re*classification).

**Conservativity** The conservativity principle states that in the absence of any declassification the security condition should revert to noninterference. We can easily model standard information-flow lattices by policies which contain sets of unguarded actors, so that for example in the two-point lattice $Low \leq High$ we would define two actors *low* and *high*, and then *Low* data would be modeled by the policy $\{\emptyset \Rightarrow low; \emptyset \Rightarrow high\}$, whereas *High* would correspond to $\{\emptyset \Rightarrow high\}$. In the presence of such unguarded policies it is straightforward to see that the notion of flow lock security reduces to the knowledge-based definition of noninterference from Askarov et al [AHSS08].

**Monotonicity of release** This principle states that adding more declassification to a "secure" program should never render it insecure. In the setting of flow locks, "adding more declassification" is naturally interpreted as *opening more locks*. A secure program which is modified to open more locks (but is otherwise unchanged) will still be secure since it is straightforward to see that the more locks are open in the lockstate at any given point in a trace, the weaker the flow lock security requirement at that point.

Formally we can state the principle of monotonicity as follows:

**Proposition 1** (Monotonicity of flow lock security)**.** *If* $\mathbf{FLS}(\Sigma, c)$ *and* $\Sigma' \supseteq \Sigma$ *then* $\mathbf{FLS}(\Sigma', c)$*.*

The proof can be found in the appendix.

**Semantic consistency** This states that the notion of security should be preserved by any semantics-preserving transformations to a program, and this is true for the semantics we define. One such example is dead code elimination. As mentioned in the introduction, lack of flow sensitivity makes security definitions sensitive to dead code. Here the definition of flow lock security can never be sensitive to dead code since it only quantifies over possible traces of a system – and these, by definition, are insensitive to dead code.

It is worth noting that semantic consistency is relative to a particular semantics; in the concrete example that we consider in the next section we assume a semantics in which the effect of assignments are directly observable (to an appropriate attacker), something which does not hold for the usual operational semantics. This is referred to as a *semantic anomaly* [SS05], and is common to many security definitions which are phrased in terms of sequences of assignments.

**Non-occlusion** The non-occlusion principle is the most vague. It tries to capture the requirement that one declassification operation should not be able to mask an arbitrary amount of future insecure information flow. In our system we can argue for non-occlusion as follows. In our definition each assignment is considered in isolation, and the presumed knowledge gained from observing an assignment is exact. Therefore any further knowledge gained by observing any future assignment must still be subject to the same constraints (modulo the knowledge gained by the earlier assignment) with respect to the lock state and policies in force at that time. Adding declassifications therefore cannot mask future unintended flows.

**Hookup Properties for Sequential Composition** In addition to the basic principles, it is useful to study composition principles (sometimes called *hook-up* properties [McC87]): when can we build secure programs from secure components.

Here we briefly consider the most basic composition principle corresponding to sequential composition. Let us suppose that we have a sequential composition operator (either directly or encodable) with the usual semantics (see the next section for example).

The termination sensitive condition has a technical problem that prevents it from composing sequentially: a program which ends in a silent loop is indistinguishable from one which terminates. This difference is revealed by composing the program with one which performs output. Termination insensitive flow lock security would consider the above composition secure, but still suffers from a problem, though for a different class of programs. A program that either silently terminates or produces one last output before termination is considered secure, since the silent termination is for all purposes equivalent to a silent loop. Composing such a program with one that performs an output again reveals the difference, and causes the previous output to be considered insecure.

To obtain secure composition, the concrete semantics used may thus not have silent termination, i.e. all programs must produce a distinguished visible

output if and only if they terminate. We say that such programs have *visible termination*. Our security definition from the previous section is agnostic as to whether visible termination is used or not.

The second minor obstacle to secure sequential composition is the lock state component. For this let us introduce Hoare-like triples $\{\Sigma\}c\{\Sigma'\}$, which state that if any computation of $c$ begins with at least locks $\Sigma$ open, on termination at least locks $\Sigma'$ will be open.

**Proposition 2.** *The following proof rule is sound, assuming the concrete semantics uses visible termination:*

$$\frac{\mathbf{FLS}(\Sigma, c_1) \qquad \{\Sigma\}c_1\{\Sigma'\} \qquad \mathbf{FLS}(\Sigma', c_2)}{\mathbf{FLS}(\Sigma, c_1; c_2)}$$

## 2.5 Enforcement: A Sound Flow Lock Type System

In this section we will illustrate our definition of flow lock security to a specific language and type system, and prove that the type system guarantees flow lock security as given by the definition in the previous section. For the sake of brevity we treat just a simple while-language, but in principle we can apply the same approach to a higher-order language and type system (which we did in our first paper [BS06b]).

### 2.5.1 Language

The simple while language presented in figure 2.1 will serve as the basis of our presentation. The only two non-standard features are statements **open** $\sigma$ and **close** $\sigma$ for manipulating the program's lock state. $\sigma$ here ranges over single locks. The notion of observable action is defined (as discussed in the previous section) as the action of assigning to a variable.

### 2.5.2 Type System

The type system we use can be found in figure 2.2. To simplify the presentation we use only int as base type for expressions, and commands have no base type, so we can restrict ourselves to only the flow locks aspects of the system. However, for convenience we use the boolean values **false** and **true** as shorthands for the value 0, and any value $v \neq 0$, respectively.

We choose to model our system as a type and effect system in the style of Almeida Matos and Boudol [AB05]. For expressions we have judgments

$$\langle n, M\rangle \Downarrow n \qquad \langle x, M\rangle \Downarrow M[x]$$

$$\frac{\langle e_1, M\rangle \Downarrow v_1 \quad \langle e_2, M\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, M\rangle \Downarrow v_1 \oplus v_2}$$

$$\langle \Sigma, \mathbf{open}\ \sigma, M\rangle \xrightarrow{\tau} \langle \Sigma \cup \{\sigma\}, \mathbf{skip}, M\rangle$$

$$\langle \Sigma, \mathbf{close}\ \sigma, M\rangle \xrightarrow{\tau} \langle \Sigma \backslash \{\sigma\}, \mathbf{skip}, M\rangle$$

$$\frac{\langle e, M\rangle \Downarrow v}{\langle \Sigma, x := e, M\rangle \xrightarrow{x(v)} \langle \Sigma, \mathbf{skip}, M[x \mapsto v]\rangle}$$

$$\frac{\langle e, M\rangle \Downarrow v \qquad v \in \{\mathbf{true}, \mathbf{false}\}}{\langle \Sigma, \mathbf{if}\ e\ \mathbf{then}\ c_{\mathbf{true}}\ \mathbf{else}\ c_{\mathbf{false}}, M\rangle \xrightarrow{\tau} \langle \Sigma, c_v, M\rangle}$$

$$\langle \Sigma, \mathbf{while}\ (e)\ c, M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbf{if}\ e\ \mathbf{then}\ c; \mathbf{while}\ (e)\ c\ \mathbf{else}\ \mathbf{skip}, M\rangle$$

$$\frac{\langle \Sigma, c_1, M\rangle \xrightarrow{\ell} \langle \Sigma', c_1', M'\rangle}{\langle \Sigma, c_1; c_2, M\rangle \xrightarrow{\ell} \langle \Sigma', c_1'; c_2, M'\rangle}$$

$$\langle \Sigma, \mathbf{skip}; c_2, M\rangle \xrightarrow{\tau} \langle \Sigma, c_2, M\rangle$$

**Figure 2.1:** Operational Semantics

of the form $\vdash e : p$ where the policy $p$, called the *read effect*, is the join of the policies on all variables whose contents are used to produce its result.

For commands the main judgments have the form $\Sigma \vdash c \rightsquigarrow p, \Sigma'$. Here $\Sigma$ is an assumption about what locks will be open before execution of $c$. The policy $p$ is the so called *write effect* of a command, which is the union of the policies on all variables whose contents might be changed when executing the command. This plays a similar role to the "PC" level in many information flow type systems. The final component $\Sigma'$ is a safe approximation (i.e. an underestimation) of the locks that will be open after execution of $c$.

Since the rules typically mention a number of different policies, we use $r$ and $w$ to range over read effect and write effect policies respectively, to simplify the presentation.

Looking more closely at some of the rules, we note that, unsurprisingly, **open** and **close** are the only commands directly affecting the lock state. In

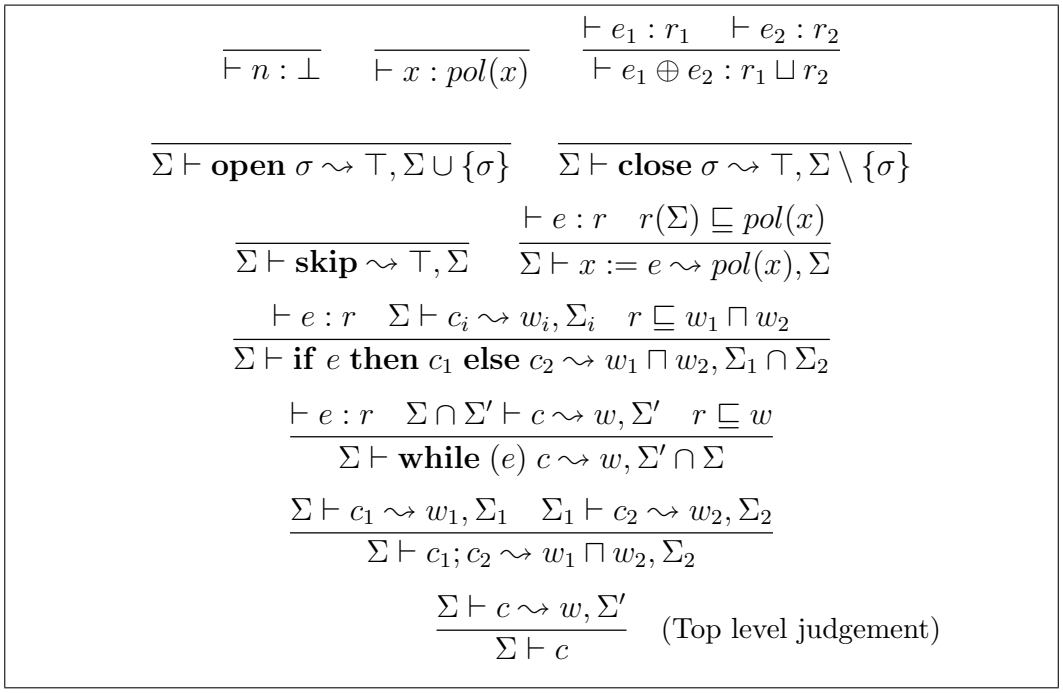



**Figure 2.2:** Flow Lock Type System

the rule for assignments, the check $r(\Sigma) \sqsubseteq pol(x)$ ensures that the assignment is valid under the current lock state $\Sigma$, thereby ruling out leaks from direct flows. In the rule for **if**, the check $r \sqsubseteq w_1 \sqcap w_2$ ensures that no indirect flows leak information about a "secret" conditional expression to "public" locations. Similarly for the test $r \sqsubseteq w$ in the **while** rule.

For the **if** rule, to compute a safe approximation to the locks that will be open it suffices to take the intersection of the resulting lock states of the branches. The **while** rule needs to use a fix point for the resulting lock state since one iteration of the loop may close locks that would then not be open in subsequent iterations.

We note that there is a natural subtyping in the lock state component of this type system. Formally

$$\Sigma \vdash c \rightsquigarrow w, \Sigma' \wedge \Delta \supseteq \Sigma \Longrightarrow \Delta \vdash c \rightsquigarrow w, \Delta' \wedge \Delta' \supseteq \Sigma'$$

This is easily proved by looking at all the uses of the lock states in the rules, and in particular noting that $\Sigma$ is covariant in $r(\Sigma) \sqsubseteq pol(x)$ in the rule for assignment.

Further, we can formalise our claim that the resulting lock state in the type system is a safe approximation of running the command as follows:

**Proposition 3** (Lockstate Safety). $\Sigma \vdash c \rightsquigarrow w, \Sigma'$ *implies* $\{\Sigma\}c\{\Sigma'\}$.

The proof of this is a straightforward induction over the typing derivations. We of course also want to prove soundness with respect to progress and preservation. We have that

**Proposition 4** (Progress). *If* $\Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $c \neq \textbf{skip}$ *then* $\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Sigma', c', M'\rangle$.

**Proposition 5** (Preservation). *If* $\Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $\langle \Sigma, c, M\rangle \xrightarrow{\ell} \langle \Sigma', c', M'\rangle$ *then* $\Sigma' \vdash c' \rightsquigarrow w', \Delta'$ *for some* $w' \sqsupseteq w$ *and* $\Delta' \supseteq \Delta$.

Proof of Progress is a straightforward case on the syntax of commands. Proof of Preservation is much more involved and can be found in the appendix.

Finally we can state the main proposition of this section, which is that the type system implies flow lock security. The type system is formulated in a termination insensitive way, in particular we allow low assignments after high loops, so that is the formulation we will prove.

**Theorem 1** (Well typed programs are flow lock secure). *If* $\Sigma \vdash c$ *then* $\mathbf{FLS_{TI}}(\Sigma, c)$.

The proof is given in the appendix.

## 2.6 Example Encodings

Many declassification ideas can be encoded using flow locks. By such an encoding we now obtain a weaker flow-sensitive semantics for the corresponding declassification mechanism.

### 2.6.1 Delimited Non-Disclosure

As a simple example let us take a recent declassification mechanism, *delimited non-disclosure* [BCR08]. In its simplest form we have variables of either *High* or *Low* security levels, and a local block-structured declassification command **declassify** $h$ **in** $c$ which allows a local weakening of the policy so that $h$ is treated as low for the computation of command $c$. This is a variable-centric variant of Almeida Matos and Boudol's non-disclosure construct [AB05].

To encode this idea using flow locks we need to use one lock $Decl_h$ per high variable $h$. Then we assign the policy $pol(\ell) = \{high, low\}$ for each low

variable $\ell$, and $pol(h) = \{high, Decl_h \Rightarrow low\}$ for each high variable $h$. The encoding of **declassify** $h$ **in** $c$ is then the obvious

$$\textbf{open}\ Decl_h; c; \textbf{close}\ Decl_h$$

For this encoding we need to assume that there are no nested declassifications over the same variable. This is not a real restriction since the inner declassification would be redundant in that case.

The semantics of delimited non-disclosure is bisimulation-based with memory resetting, so suffers from flow insensitivity (see the example in Section 2.3). We conjecture that our encoding gives a strictly weaker semantics, but that encoded programs typable in our simple type system are also typable in the system given in Barthe et al [BCR08]. This is because our type system is too simple to take advantage of flow sensitivity.

### 2.6.2 Gradual Release

A more interesting example is provided by the *Gradual Release* property from Askarov and Sabelfeld [AS07]. This is interesting because the style of definition used there was the inspiration for our approach. Surprisingly we are able to show that, when specialised to the case of simple declassification, our definition coincides exactly with gradual release.

We will begin by presenting their core operational semantics, as well as the Gradual Release security requirement for programs. We will then present a simple encoding of their language using flow locks, and show that for the class of flow locks programs conforming to the encoding, the two operational semantics and security requirements are equivalent. We will also show that their type system is equivalent to the type system given for the example language in Section 2.5, for that same class of programs.

The language used by Askarov and Sabelfeld [AS07] is a simple while language similar to the one presented in section 2.5. It uses a simple two-level lattice $\mathcal{L} = \{\text{Low}, \text{High}\}$ with $\text{Low} \sqsubseteq \text{High}$ and $\text{High} \not\sqsubseteq \text{Low}$. As expected data may flow freely from locations marked with Low to locations marked with High, but not the other way around. The special **declassify** command allows a program to leak data from High to Low.

The relevant parts of the operational semantics for this language can be found in Figure 2.3. There should be no surprises apart from the outputs arising from assignments and declassifications. These are labeled differently – normal assignments to variables marked with Low cause outputs of the form $x(v)$ whereas declassifications output so called *release events* denoted by $r : x(v)$. Assignments to variables marked with High do not yield any outputs at all.

$$\frac{\langle M, e\rangle \Downarrow n \quad pol(x) \neq \text{Low}}{\langle M, x := e\rangle \rightarrow \langle M[x \mapsto v], \textbf{skip}\rangle}$$

$$\frac{\langle M, e\rangle \Downarrow n \quad pol(x) = \text{Low}}{\langle M, x := e\rangle \xrightarrow{x(v)} \langle M[x \mapsto v], \textbf{skip}\rangle}$$

$$\frac{\langle M, e\rangle \Downarrow n}{\langle M, x := \textbf{declassify}(e)\rangle \xrightarrow{r:x(v)} \langle M[x \mapsto v], \textbf{skip}\rangle}$$

**Figure 2.3:** Operational semantics for Gradual Release

The set of all possible low event sequences of a program is defined as follows:

**Definition 11** (Low event sequences)**.** The set of all possible low event sequences that program $c$ may generate starting from a low memory $L$ is

$$GRRun(c, L) = \{\vec{u} \mid M =_{\text{Low}} L, \langle M, c\rangle \xRightarrow{\vec{u}} \langle M', c'\rangle\}$$

where $=_{\text{Low}}$ is equivalence on the low part of the memory.

For a program to satisfy Gradual Release, it needs to fulfill the following property[4]:

**Definition 12** (Gradual Release)**.** A command $c$ satisfies *Gradual Release*, written $GR(c)$, if for all low projections of memories $L$, and all pairs of sequences $\vec{u}u, \vec{u}u' \in GRRun(c, L)$, we have

$$k(c, L, \vec{u}u) = k(c, L, \vec{u}u')$$

**Flow Locks Encoding** The language displayed here is as noted already very similar to that shown in Section 2.5 and the encoding is straightforward as previously described. We define an encoding function $\widehat{\cdot}$ over commands, and policies etc.

First we need to represent the security levels High and Low. As before we introduce two actors: *low* is only allowed to see public (Low) data, while *high* is allowed to see any data. We also introduce a lock *Decl* to handle declassification. We can then encode the two levels as $\widehat{\text{High}} = \{high; Decl \Rightarrow low\}$

[4]We take the liberty of presenting the definitions from Askarov and Sabelfeld [AS07] in a style that more closely resembles those which we have used for our own definitions. Our presentation is not different in any substantial way.

and $\widehat{\text{Low}} = \{high; low\}$. We have $\widehat{\text{Low}} \sqsubseteq \widehat{\text{High}}$ and $\widehat{\text{High}} \not\sqsubseteq \widehat{\text{Low}}$, as expected. We extend the encoding to variables ($\widehat{x}$) and memories ($\widehat{M}$) in the obvious way, by encoding all policies involved.

Secondly, we need to encode declassification. As previously the command

$$x := \textbf{declassify}(e)$$

is represented with the sequence of commands

$$\textbf{open}\ Decl; x := e; \textbf{close}\ Decl$$

And that is all we need.

**Equivalence** Our main goal here is to show that our encoding of the Gradual Release primitives leads to a system that is equivalent to the original Gradual Release system presented by Askarov and Sabelfeld [AS07]. In particular, we want to show that a program will be deemed secure according to Gradual Release if and only if its encoding is deemed flow lock secure.

$$GR(c) \equiv \mathbf{FLS_{TI}}(\emptyset, \widehat{c})$$

To do this, we first note that on the flow locks side the only possible lockstates at any point in the program are $\mathcal{P}(Locks) = \{\emptyset, \{Decl\}\}$, and the only actors are $Actors = \{high, low\}$. Further we note that for all attackers $A \in Actors \times \mathcal{P}(Locks)$, the only attacker that would not have perfect knowledge of the memory at all times is $A = (low, \emptyset)$. We can then specialise the definition of flow lock security, to say that an encoded program $\widehat{c}$ is termination insensitive flow lock secure iff for attacker $A = (low, \emptyset)$, for all $A$-low memories $\widehat{L}$, and all pairs of runs $(\vec{u}u, \emptyset), (\vec{u}u', \Omega) \in Run_A(\emptyset, \widehat{c}, \widehat{L})$ we have that

$$k_A(\vec{u}u, \widehat{c}, \widehat{L}) = k_A(\vec{u}u', \widehat{c}, \widehat{L})$$

Next we note that we have a simple correspondence between the definitions of runs.

**Lemma 1** (Correspondence between runs)**.** *If* $(\vec{u}u) \in GRRun(c, L)$ *then* $(\vec{u}w, \Omega) \in Run_A(\emptyset, \widehat{c}, \widehat{L})$ *for* $A = (low, \emptyset)$*, and further* $\Omega = \{Decl\}$ *iff* $u$ *is a release event, otherwise* $\Omega = \emptyset$*.*

The proof of this is a straightforward inspection of $c$.

Applying lemma 1 to our specialised version of flow lock security above, we end up with exactly definition 12, which is what we wanted to prove.

**Type system equivalence** We can also show that the type system presented by Askarov and Sabelfeld [AS07] is equivalent to the type system presented in Section 2.5. The typing judgments and rules for expressions in the gradual release system are identical to those in our type system. For commands, we have that

$$\vdash_{\text{GR}} c \rightsquigarrow w \iff \emptyset \vdash \widehat{c} \rightsquigarrow \widehat{w}, \emptyset$$

This is trivial to show for all commands except for assignments and declassifications.

For assignments the rule for the Gradual Release system states that

$$\frac{\vdash_{\text{GR}} e : r \quad r \sqsubseteq pol(x)}{\vdash_{\text{GR}} x := e \rightsquigarrow pol(x)}$$

and we have a direct correspondence with the type rule for assignments from Section 2.5, specialised to encoded commands:

$$\frac{\vdash e : \widehat{r} \quad \widehat{r} \sqsubseteq \widehat{pol(x)}}{\emptyset \vdash x := e \rightsquigarrow \widehat{pol(x)}, \emptyset}$$

For declassification the rule from Gradual Release is simply

$$\frac{\vdash_{\text{GR}} e : r}{\vdash_{\text{GR}} x := \mathbf{declassify}(e) \rightsquigarrow pol(x)}$$

i.e. no constraints on the respective security labels of $e$ and $x$. For the encoded equivalent,

$$\mathbf{open}\ Decl; x := e; \mathbf{close}\ Decl$$

we can simply construct the derivation and everything is trivially typable, with the exception of the constraint $r(\{Decl\}) \sqsubseteq pol(x)$ arising from the assignment in the middle. Since we know from the domain that $r$ is either $\{high; low\}$ or $\{high; Decl \Rightarrow low\}$, we have that $r(\{Decl\}) = \{high; low\} = \widehat{\text{Low}}$. Since for all $l$, $\widehat{\text{Low}} \sqsubseteq l$, the constraint $\widehat{\text{Low}} \sqsubseteq pol(x)$ is always fulfilled, and we are done.

**Discussion** What we hope to show with this encoding is that this could have been a feasible (not to say easy) way to prove properties about Gradual Release. The proofs here that Gradual Release is a specialisation of Flow Locks are much less involved than the proofs in the Gradual Release paper, even though those are quite simple to begin with.

Gradual Release is a special case in that it is already flow sensitive, so we get an exact equivalence between the original semantics and the flow locks induced one. We could not get such a correspondence with a flow insensitive system. However, we argue that most other systems are not inherently flow insensitive, and that giving a flow sensitive semantics to them via a flow locks encoding is not only feasible, but also beneficial since it makes it easier to relate various semantics and enforcement mechanisms.

Reasoning about flow locks is greatly simplified by the new form of semantics. But what we have not done in these examples is take advantage of the fact that the semantic condition is not only simpler but also more liberal: in fact the type system we have presented is very similar to that which we originally verified against a flow *insensitive* semantics in our first paper [BS06b]. Flow sensitivity would be useful in cases where the type system also needs to track properties of values – for example if we wanted to extend the typings to additionally verify that openings of locks only occurred in specific states, or released specific parts of some data (c.f. Banerjee et al [BNR08]). Any resetting-style semantics would not be able to track such properties through a computation.

### 2.6.3 More encodings

**Intransitive Noninterference** Flow locks represent a lower level abstraction than lattice-based information flow models in the sense that the lattice ordering is not "built in" but must be represented explicitly. One advantage of such a lower level view is that it can also represent *intransitive noninterference* policies [Rus92, Pin95] — i.e. ones in which the flow relation is intentionally not transitive. Since intransitive policies are the default case for flow locks, it is straightforward to represent simple language-based intransitive policies such as the one described by Mantel and Sands [MS04].

**Noninterference Until Declassification** Chong and Myers [CM04] introduce a class of temporal declassification policies. This is achieved by annotating variables with types of the form $k_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_n}{\rightsquigarrow} \underline{k_n}$, which intuitively means that a variable with such an annotation may be successively declassified to the levels $k_1, \ldots, k_n$, and that the conditions $c_1, \ldots, c_n$ will hold at the execution of the corresponding declassification points. The exact nature of the conditions are left unspecified, and it is assumed in the type system that these conditions are verified at certain key program points by some external tool.

We can achieve a similar effect fairly naturally using flow locks, where we would use a distinct lock $C_i$ for each condition $c_i$. One should then insert **open** $C_i$ constructs in the program at points where the intended declassification takes place, and verify (with an external tool) that the corresponding condition $c_i$ does indeed hold at these points, and that lock $C_{i-1}$ has been opened (we assume that locks are never closed in this encoding). The policy above could then be represented as

$$\{k_0; \{C_1\} \Rightarrow k_1; \cdots ; \{C_1, \ldots, C_n\} \Rightarrow k_n\}.$$

**Robust Declassification** Information flow may be used to verify integrity properties, to ensure that untrusted (low integrity) data does not influence the values of trusted (high integrity) data. Since flow lock policies are neutral with respect to whether we are dealing with confidentiality or integrity properties it is no problem to add such integrity policies to data, and we can easily have clauses for integrity and confidentiality in the same policy. The interesting case, however, is the interaction between confidentiality and integrity in the presence of dynamic policies.

Zdancewic and Myers [ZM01] introduced the concept of *robust declassification* to characterise the property that an attacker (who controls low integrity data) cannot influence what is declassified. This guarantees that the attacker cannot manipulate the amount of information which is released through declassification.

In the setting of flow lock policies, "declassification" can be thought of as the process of opening locks, since whenever a lock is opened more flows are enabled. Thus we can interpret robust declassification as the question of whether low integrity data can influence the decision to open locks. [5]

One possible way of enforcing robust declassification using flow locks is to observe the following: since we cannot perform any computation with locks, the only way that an open operation can be influenced by low integrity data is via indirect information flow from low integrity data. Suppose that our policies use an indexed set of locks $\sigma_i, i \in I$ to control confidentiality. These are unguarded (i.e. we ignore *endorsement*). Let us assume that in addition to the actors of the system we have the pseudo-actor *trusted* used to track integrity information, just as we did in Section 2.2.

Since flow locks themselves do not have policies, since the information they carry cannot be used to influence values at runtime, we need to use an auxiliary trick to encode the flow from data to locks. In order to prevent

[5] If we also take the view from Myers et al [MSZ04], then we extend this concept with the requirement that we should not be able to declassify low integrity data

indirect flow from low integrity data to the opening of locks, we will log each use of an open operation by writing to a variable *log*. An obvious way to enforce this is to define a "robust" version of open:

$$\textbf{ropen}\ \sigma_i \equiv \textbf{open}\ \sigma_i; log := i$$

Now we give *log* the policy $\{trusted\}$. This ensures that the assignment is always safe from a confidentiality perspective (since normal actors can never read it anyway), and that the open operation can never have taken place in a low integrity context (since otherwise the assignment would cause information to flow from untrusted to trusted data). Finally, to additionally prevent the declassification of low integrity data we can syntactically enforce that lock-guarded policies are only used on high integrity data.

In the next chapter we allow information flow from locks to values, requiring locks to have policies too, which makes the encoding of robust declassification even more straightforward.

**The Decentralised Label Model** In the Decentralised Label Model (DLM) [ML97, ML98, ML00], data is said to be *owned* by a set of principals. These principals may allow other principals to read the data, and the effective reader set is those principals that all owners agree may read the data. Allowing a new reader roughly corresponds to declassification, and we can model it similarly. The DLM also defines a global principal hierarchy, where one principal may allow another principal to *act for* it, which means it may read all the same things. This is very similar in spirit to introducing a new flow in the system by Almeida Matos and Boudol, including transitivity, and we can model it in the same way. Apart from clauses for declassification and hierarchic flows, the policies must also include clauses for the combination of the two, e.g. $A$ can read the data if $B$ owns it, has declassified it for $C$ to read it, and $A$ acts for $C$.

A common extension of the DLM [ZM01, TZ05, TZ04] deals with integrity and trust. The interesting part for us is the integration with the principal hierarchy, where if $A$ trusts some data and $A$ acts for $B$, then $B$ also trusts that data. This can be modeled as the reverse of the normal clauses for transitive flows, and the clauses will be very similar to those for forward flows.

The complete general policy for a DLM variable encoded with flow locks would be fairly large and awkward due to the lack of a number of abstraction mechanisms. That is the topic of the next chapter.

# Chapter 3

# Paralocks

## 3.1 Introduction

As noted in the introduction chapter, issues of software security can be crudely categorised into three broad domains:

- *Access control* deals with security at the end points of a system, to verify that an entity is allowed to access the system, and to what extent.

- *Information flow control* deals with security *inside* a system, between the end points, to ensure that data in the system is handled in a way that agrees with the security policy of the system. This is the domain that is most interesting from a programming language point of view, since it deals with security during execution.

- *Encryption* deals with security *outside* a system, to ensure that data can be protected even outside the trusted system environment.

The problems involved in research on encryption are quite different from the other two domains, but unsurprisingly there are many similarities between problems that arise in the access control and information flow control domains. In particular, problems regarding policy specification and modeling of principal actors are quite similar, much due to the fact that these issues are not purely technical, but rather relate to the interface between the system and its users (implementor, admins). Thus, many ideas relevant in one domain are equally applicable to the other, at least on a high level.

In the access control domain there exists plenty of research regarding policy specification mechanisms. Such mechanisms have traditionally been categorised into two separate groups: Mandatory (or static) access control

(MAC), where an outside administrator assigns static privileges to principals, and Discretionary access control (DAC), where principals themselves can grant and revoke privileges to and from other principals. A later addition to the family of models is Role-based access control (RBAC) [SCFY96], which has become very popular and has seen wide-spread adoption both commercially and academically.

On the information flow control side, there has been far less focus on policy specification. We surmise that this has a very natural cause. In access control, which deals with the interfaces to a system, policy specification is the one core issue and a prerequisite for any further aspects of security. Information flow control on the other hand is more naturally focused towards issues of semantic security with respect to a policy, and most research in the domain has been devoted in that direction.

Papers on information flow control issues typically fall into one of two categories where the policy mechanism used is concerned. In the first category we find those that use a simple model built around a lattice of principals or sets of principals, going back to Denning's early ground-breaking work. The other category is the research that builds on the Decentralised Label Model (DLM), which is today something of a flagship of information flow control through its implementation in Jif. These two categories can somewhat crudely be said to correspond to the MAC (static Denning-style lattice) and DAC (decentralised and discretionary) models.

Interestingly and perhaps surprisingly there has been almost no work on marrying a fundamentally role-based model to information flow control (the exceptions being Swamy et al [SHTZ06, BWW08] which are discussed further in chapter 5), despite the massive attention RBAC has received in the access control domain, both commercially and academically.

In this chapter we present Paralocks, an extension to our flow locks language from the previous chapter. Paralocks extends flow locks with the ability to express policies modeling roles (in the style of RBAC) and run-time principals.

The extension (*para*meterised *locks*) turns out to provide much more than just the ability to model roles: we show (section 3.6.2) how relations such as delegation in discretionary access control can be represented by policies, and use this to give a sound and complete encoding of the Decentralised Label Model (DLM) [ML97].

Unlike the DLM we also provide an information flow semantics for Paralocks. This defines what it means for a program (whose state components are labeled with policies) to be secure.

As an illustration of how Paralocks can be integrated into a programming language we give an example of a small programming language with a Par-

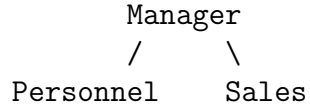
alocks type system for which we show that well-typed programs satisfy the semantic information flow condition.

Finally we outline a logically natural extension to the Paralocks policy language to include recursively specified locks (DATALOG rules).

Our aim with Paralocks is really two-fold. On the one hand we present the policy specification language itself, as well-suited for specifying information flow policies in practice. Paralocks extends flow locks to naturally models roles, but also actors, groups and general relationships in a simple and structured way. On the other hand, Paralocks is a very general framework for information flow control. This aspect lets us use Paralocks to reason about and give meaning to other mechanisms both current and future. Paralocks thus serves as a platform that can greatly simplify further research into various aspects of information flow control, such as specialised policy specification languages, and the relationship between information flow control and programming language design.

## 3.2 Roles and Information Flow

Roles are a natural concept in an organisational structure and are just as naturally tied to information flow controls as to access control. Consider a department consisting of managers, personnel and sales. These roles form a hierarchy as illustrated in the Hasse diagram below:




In role-based access control each role represents a set of users (later we will use the neutral term *actors*) endowed with a set of permissions. The hierarchy illustrated in the figure (roles + hierarchies are referred to as RBAC1 in the RBAC96 model [FSG+01]) represents the intention that the permissions granted to higher roles subsume those granted to lower roles.

Let us suppose that we take an information flow perspective on roles and we assume data is labeled with a role, representing the permission to gain information about that data. Then role-based information flow control would simply be the constraint that information may only flow upwards in the hierarchy. This is simply the Denning lattice-based model [Den76] with a relaxation on the requirements that the hierarchy forms a complete lattice.

In this setting the assignment of users to roles is of little direct concern from an information flow perspective, since users do not possess their own data, and are defined purely by the roles to which they are assigned. Inputs

and outputs of the system would then be bound to roles, and some external mechanism would mediate the connection between roles and users.

However, if we admit the possibility of personal data then the information flow perspective becomes considerably richer. For example, if we had I/O channels directly to users then we would have an information flow problem with a dynamic policy: information flows to and from a given actor would depend on her current role.

Consider another scenario involving personal data, a twist on the example we used in the previous chapter: an auction site managing sealed-bid auctions for an *a priori* unspecified number of users. In such a scenario the roles of seller and bidder, respectively, immediately spring to mind. Other constraints on the auction influence the intended information flows:

- the seller can set a reserve price which is initially only visible to the seller;
- bidders provide sealed bids and can see their own bid but cannot see each others' bids;
- bidders learn of the winning bid, but only at the end of the auction;
- if the reserve price is not met then there is no winning bid;
- sellers cannot also be bidders for the same item.

In summary, to verify that code managing such auctions is well behaved raises a number of general challenges from an information flow perspective, some of which our flow locks from the previous chapter cannot handle:

1. We need to model dynamic actors – actors whose concrete identity is not known or may not exist until runtime.
2. The data associated with a role (e.g. the bids) belongs to the actor and not the role (because bidders should not be able to see all bids - only their own bid).
3. Permissions associated with roles are assigned dynamically (in this example, the ability to read a winning bid is only granted after the auction is complete).
4. Declassification is required: the winning bid (or its absence) provides partial information about the secret reserve price of the seller.
5. We must be able to impose role constraints (a la RBAC2) to ensure that the seller cannot become a bidder on the same item.

In the next section we extend flow locks into a language aimed at meeting these challenges. Our extension is motivated by the addition of *roles*. However, as we will show in a later section (3.6.2), the extension turns out to provide considerably more than just the ability to represent roles.

## 3.3 Flow Locks and Roles

Consider a simplified form of the auction example in which we have two known buyers $B_1$ and $B_2$ and a single seller $S$ and where the bidders may see each other's bids once they have placed their own. We associate two locks, $bid_1$ and $bid_2$ with the placing of bids by $B_1$ and $B_2$ respectively; $bid_i$ will be assumed to become true once $B_i$ has placed his bid. Then the policy for $B_1$'s bid is

$$\{S; B_1; bid_2 \Rightarrow B_2\}.$$

This says that the bid of $B_1$ may flow to $S$ and $B_1$ unconditionally, and may flow to $B_2$ only when $B_2$ has placed a bid (as modeled by lock $bid_2$).

As can be seen, when using simple flow locks, all actors must be known statically and enumerated in policies, and we need locks that are specific to each such actor.

Consider a further example represented in Figure 3.1 which depicts three Denning-style information-flow lattices.

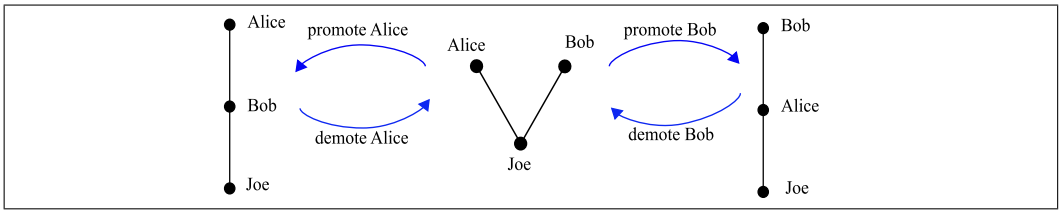



**Figure 3.1:** Example Dynamic Policy

In the leftmost lattice Alice is the top element. While Alice is "boss" all information may flow to her. If she is demoted, however, then the information flow lattice changes to the central figure. From there either Bob or Alice can be promoted to be the boss. Let us consider how to encode this intended scenario with flow locks. To represent this dynamic flow policy we begin, not surprisingly, by assuming three actors: *Alice*, *Bob*, and *Joe*. To model the transitions between policies we use two locks: *promoteA* and *promoteB*. The events of promotion and demotion are modeled by the respective opening and closing of these locks. When *promoteA* is open then Alice is boss. Closing *promoteA* (respectively, *promoteB*) corresponds to demoting Alice (resp. Bob).

To complete the picture we need to describe the corresponding policies for the data to be associated with Alice, Bob, and Joe. Joe is the simplest case, and his data has policy $\{Joe; Alice; Bob\}$ – i.e. it is readable by everyone at all times. Alice's data has policy $\{Alice; promoteB \Rightarrow Bob\}$ and Bob has the symmetric policy $\{Bob; promoteA \Rightarrow Alice\}$. For Bob this means that his data is readable by Alice only when Alice has been promoted. Note that if both locks are open then we have a situation not modeled in the figure: Alice and Bob become equivalent from an information flow perspective. If we want to rule this out we cannot do so using the policy on data, instead we must enforce this via an invariant property of the locks themselves.

Again, using flow locks we can only reason about actors that are statically known, and duplicate locks across actors and policies. To write these examples in a convenient and flexible way, we need the concept of *roles*.

### 3.3.1 Modeling Roles

A naive approach to supporting roles with Flow locks could be to simply let the actors *be* the roles, and not have conventional actors at all. This would work with no changes to the current policy language, however as our examples in the previous section have shown, we often need to reason about both roles (or groups) *and* individual actors.

Thus we need a different approach to roles that retains the notion of an actor. Looking at what it means for some data to be accessible to a role $R$, the natural interpretation is that an actor may gain information about that data if the actor is a member of $R$. How do we express this as a flow lock policy? We need a lock that captures the condition that "$a$ is a member of role $R$", which we henceforth write $R(a)$. But clearly the policy we want is not just for some specific actor $a$, but rather any actor $x$ for which $R(x)$ holds. Logically we could easily write this as $\forall x.\, R(x) \implies x$. A role thus has a natural representation as a lock *family*, parametrised by actors.

To achieve this we introduce two separate – though synergistic – extensions to the basic formulation of flow locks:

**Parameterised Locks** Locks which are parameterised over actors represent role membership. For example the role *Seller* is represented as parameterised lock family, so if $a$ is an actor then $Seller(a)$ is a lock which models $a$ being a member of the seller role. Data labeled with the policy $\{Seller(a) \Rightarrow a\}$ is permitted to flow to $a$ providing that $a$ is a seller.

**Actor Polymorphism** To make parameterised locks practically useful we also need to be able to quantify over *all* actors, so that we could instead

write the policy as $\{\forall x.\, Seller(x) \Rightarrow x\}$, meaning that data labeled with this policy may flow to *any* seller.

With this interpretation of roles, and these extensions to the policy specification language, we can easily formulate the policies from the examples in the previous section using flow locks. Let us then return to the challenges offered by those examples:

1. Actors whose concrete identity is not known until run-time can be handled by policies with actor polymorphism. As a simple example, the policy $\{\forall x.\, x\}$ is the most liberal policy, permitting its data to flow to any actor at all times. This does not require us to know the identity of all actors at policy creation time (as would be required using the original basic flow locks mechanism).

2. Fine grained policies at the level of individual actors combine easily with roles. For example, suppose we wish to generalise the scenario in Figure 3.1 to an organisation of 1000 employees – or a situation with an unknown number. Here we must combine a role (the boss) with the requirement that non-bosses cannot obtain information from each other (with the exception of Joe). The data of Joe would have policy $\forall x.\, x$ – it can flow to anyone. The data for any other individual $a$ would have the policy $\{a; \forall x.\, Boss(x) \Rightarrow x\}$, which means that data labeled with this policy can flow to $a$ and anyone who is a boss (at the time of the flow).

3. Permissions associated with roles are assigned dynamically by using standard (non-parameterised) locks. For example, the largest bid might be stored in a variable with policy

$$\{\forall x.\, \{AuctionClosed, Bidder(x)\} \Rightarrow x\}$$

   where the vanilla lock *AuctionClosed* represents the property that the auction is complete and the reserve has been met. So in effect *AuctionClosed* represents the condition under which the *Bidder* role is assigned the permission to learn about the winning bid.

4. Declassification is inherited from the standard flow locks model. For example, the reserve price is available to the seller, but is declassified to bidders providing there is a winning bid:

$$\{Seller; \forall x.\, \{AuctionClosed, Bidder(x)\} \Rightarrow x\}$$

5. Role constraints – here the requirement that e.g. there is a single seller, and that seller and buyer cannot be the same actor – can be established by runtime invariants for flow locks. These can either be verified statically or enforced dynamically using a runtime representation of locks. In Section 3.7 we describe an extension which permits certain constraints on roles to be specified as part of the policy.

### 3.3.2 The Paralocks Policy Language

Now we can summarise the policy language, and define its lattice structure. In summary, the policy language generalises flow locks policies with actor-parametrised locks, hence the name: *Paralocks*. The ordering on policies is based on a straightforward and natural logical interpretation of policies.

**Definition 13** (Paralock Policies)**.**

- Policies are built from *actor identifiers*, ranged over by $a$, $b$, etc. and *parameterised locks*, ranged over by $\sigma$, $\sigma'$ etc. Each parameterised lock has a fixed arity, $\text{arity}(\sigma) \geq 0$.
- A *lock* is a term $\sigma(a_1, \ldots, a_n)$, where $\text{arity}(\sigma) = n$. Let $\Sigma$, $\Sigma'$ range over sets of locks.
- A *clause* $c$ is a term of the form $\forall a_1, \ldots, a_n.\, \Sigma \Rightarrow a$.
- A *policy* $p$ is a set of clauses written $\{\, c_1; \ldots; c_n\}$.

We have already adopted a number of syntactic abbreviations in earlier examples: we write just $\sigma$ instead of $\sigma()$ in the case that $\text{arity}(\sigma) = 0$. Similarly we drop the quantifier on clauses when there are no quantified variables. When the lock set $\Sigma$ in a clause is empty, as in $\forall a_1, \ldots, a_n.\, \emptyset \Rightarrow a$ we write $\forall a_1, \ldots, a_n.\, a$. We will routinely write $\vec{a}$ to denote some sequence $a_1, \ldots, a_n$. Such a sequence will be treated as a set $\{a_1, \ldots, a_n\}$ when the context permits us to do so without ambiguity.

Policies have a natural reading as conjunctions of definite first-order Horn clauses. Each clause

$$\forall a_1, \ldots, a_n.\, \{\, \sigma_1(\vec{b}_1); \ldots; \sigma_m(\vec{b}_m)\} \Rightarrow a$$

can be read as the Horn clause

$$\forall a_1, \ldots, a_n.\, (\sigma_1(\vec{b}_1) \wedge \cdots \wedge \sigma_m(\vec{b}_m)) \Rightarrow \mathit{Flow}(a)$$

where *Flow* is a single unary predicate disjoint from the parameterised locks, representing the "may flow to" property.

Using this logical interpretation we obtain a natural lattice structure on policies, where the policy ordering ($\sqsubseteq$) on individual clauses is just logical entailment. Specifically, we define $p \sqsubseteq q$ whenever $p$, viewed as a first order formula, entails $q$. We will write $p \models q$ to denote this logical interpretation.

Following this natural interpretation we have the following definition:

**Definition 14** (Policy ordering)**.** Policy $p_1$ is *less restrictive than* policy $p_2$, written $p_1 \sqsubseteq p_2$, if $\forall c_2 \in p_2. \exists c_1 \in p_1. c_1 \sqsubseteq c_2$, where the ordering $\sqsubseteq$ on clauses is defined to be the least partial order (reflexive and transitive relation) satisfying the following:

- if $c_1$ and $c_2$ are equal up to (i) capture-free renaming of $\forall$-bound actors (ii) reordering of quantified actors and (iii) deletion of $\forall$-bound actors not occurring in the body of the clause, then $c_1 \sqsubseteq c_2$;
- $\forall a_1, ..., a_n. \Sigma_1 \Rightarrow b \sqsubseteq \forall a_1, ..., a_n. \Sigma_2 \Rightarrow b$ if $\Sigma_1 \subseteq \Sigma_2$.
- $\forall a_0, a_1, ..., a_n. \Sigma \Rightarrow a \sqsubseteq \forall a_1, ..., a_n. ((\Sigma \Rightarrow a)[a_0 := b])$, where $[a_0 := b]$ denotes the unconstrained substitution of $b$ for $a_0$.

We do not present a formal proof that this corresponds to the logical interpretation (in fact we did not spot the connection directly), but we note that clauses are equivalent to so-called *conjunctive queries* [CM77], and a policy thus a union of conjunctive queries. The ordering on clauses defined above can be seen as a construction of a *containment mapping* [Ull90]. The fact that $\forall c_2 \in p_2. \exists c_1 \in p_1. c_1 \sqsubseteq c_2$ is necessary and sufficient to check logical entailment of unions of conjunctive queries was established by Sagiv and Yannakakis [SY80].

At any point during program execution, the permitted flows will depend on the locks which are open at that point. To determine whether $p \sqsubseteq q$ in the context of some open locks $\Sigma$, we check the logical implication $\Sigma \wedge p \models q$. In the type system given in Section 3.5 we implement this check via the *specialisation* of policy $p$ to a lock state $\Sigma$, written $p(\Sigma)$; we then check that $p(\Sigma) \sqsubseteq q$.

The meet operation on policies is simple to define as it corresponds exactly to conjunction of (sets of) Horn clauses. In our language, that means taking the union of the clauses of two policies, i.e.

**Definition 15** (GLB)**.** $p_1 \sqcap p_2 = p_1 \cup p_2$

The join operation however is more tricky. Logically it corresponds to a best approximation of disjunction of Horn clauses, since in general (sets of) Horn clauses are not closed under disjunction. I.e. $p \sqcup q$ is the least policy such that $p \vee q \models p \sqcup q$. We can define the join directly as follows:

**Definition 16** (LUB)**.** In the following it is convenient to partition actor variables in to $\forall$-bound variables ranged over by $x$, $\vec{y}$, and free actor variables (i.e. actor constants) ranged over by $a$ and $b$. We write $\Sigma \Rightarrow b$ to denote the policy $\forall \vec{y}.\, \Sigma \Rightarrow b$ where $\vec{y}$ are the $\forall$-bound variables of $\Sigma \Rightarrow b$.

Let $p$ and $q$ be policies. We will assume, without loss of generality, that all $\forall$-bound variables appearing in the head of any clause are named $x$, and that any other $\forall$-bound variables in any clause from $p$ are distinct from the $\forall$-bound variables of $q$.

Then we define

$$\begin{aligned} p \sqcup q = &\{\Sigma_p \cup \Sigma_q \Rightarrow x \mid \Sigma_p \Rightarrow x \in p; \Sigma_q \Rightarrow x \in q\} \\ &\cup \{\Sigma_p \cup \Sigma_q \Rightarrow a \mid \Sigma_p \Rightarrow a \in p; \Sigma_q \Rightarrow a \in q\} \\ &\cup \{\Sigma_p \cup (\Sigma_q[x := a]) \Rightarrow a \mid \Sigma_p \Rightarrow a \in p; \Sigma_q \Rightarrow x \in q\} \\ &\cup \{(\Sigma_p[x := a]) \cup \Sigma_q \Rightarrow a \mid \Sigma_p \Rightarrow x \in p; \Sigma_q \Rightarrow a \in q\} \end{aligned}$$

It can be shown that the set of Paralocks policies (quotiented by the equivalence relation generated from $\sqsubseteq$) form a complete lattice. We will not go into the proof here, but simply note the least (most liberal) policy $\bot = \{\forall x.\, x\}$ and the greatest (most restrictive) policy $\top = \{\ \}$, which will be needed later.

### 3.3.3 Beyond Roles

Using actor-indexed lock families we have shown how we can model roles along-side specific actors in a natural logical setting, and how the two can co-exist in the same program. Next we will show how, using a natural generalisation, we can model policies where information flow can depend on *relations* between actors. Such relations are useful in the description of a decentralised discretionary security model.

The core components of a decentralised discretionary model is the concept of *ownership*, and an *acts-for* relationship (sometimes referred to as *delegation* or a *speaks-for* relation [LABW91]), where an actor $a$ who acts for $b$ enjoys the same rights as $b$. In particular if actor $a$ owns some data then $b$ has full access to that data if $b$ acts for $a$. The condition under which $b$ may access the data is thus that "$b$ acts for $a$". Logically this is easily modeled with a binary relationship between actors, which in the flow locks setting would naturally correspond to a lock family with *two* parameters. The policy mentioned here could then be written as $\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x\}$.

Going from one to two parameters, or indeed to $n$-ary lock families, is a straightforward generalisation. There are no additional technical difficulties involved, and we already have the mechanics for quantification in place. (We

have no immediate examples of lock families with more than two parameters, but see no reason to exclude them.)

## 3.4 Paralocks Security

In the previous chapter we developed a simple and accurate context-sensitive security model for flow locks based on understanding when an attacker's knowledge about initial data values is permitted to increase, developed as a generalisation of the simple *gradual release* definition [AS07].

The semantic model developed in this section is an extension of the simple flow locks model. The difference is that we must handle both runtime actor allocation and runtime querying of the lock state, both of which may be sources of information flow.

### 3.4.1 Computation Model

We assume an imperative computation model – a labeled transition system – involving commands and states, but the definition is otherwise not specific to a particular programming language. We assume transitions of the form $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$ where $c$ is a program and $S$ is the program state. We assume that the semantics signals any flow of information, i.e. changes to the state, using labels $l$, where $l$ is either a distinguished silent output $\tau$ (when there is no state update), or a value $u$ corresponding to the value of the updated part of the state. So for example a simple assignment $x := 42$ would generate a $\xrightarrow{x[42]}$ transition. We further assume that the state includes at least the following three components:

- A memory, i.e. a mapping from locations to the values they contain. We denote the memory of state $S$ by $\mathbf{Mem}(S)$, and range over memories using variables $M, N$.

- A lock state, which is the set of all locks currently open. We denote the lock state of state $S$ by $\mathbf{LS}(S)$, and use $\boldsymbol{\Sigma}, \boldsymbol{\Delta}$ to range over lock states.

- An actor mapping, keeping track of the concrete run-time representation of actors that the actor variables in the program represent. We denote the actor mapping part of state $S$ by $\mathbf{Act}(S)$ use $\boldsymbol{\Lambda}$ to range over actor mappings.

Just as with program variables, actors have concrete representations at runtime, which differ from their representations in the program code. This is so we can handle e.g. dynamic creation of actors in a loop, where the same actor variable name is reused for a new actor each time around the loop. We call the runtime representations *concrete actors*, as opposed to the *abstract actors* (actor identifiers) found in the program code and policies.

As a consequence, since locks can take actors as arguments, at runtime locks will be parametrised by concrete actor representations. We refer to a lock with concrete actor parameters as a *concrete lock*. The lock state component of the state consists of the set of concrete locks currently open.

For both actors and lock sets we adopt the convention to use bold face identifiers when denoting concrete entities. For instance, $\Sigma$ would represent a set of abstract locks in e.g. a policy stated in the program, while $\mathbf{\Sigma}$ ranges over sets of concrete locks. For actors, $\Lambda$ ranges over sets of abstract actors, while (with a slight abuse of our convention) $\mathbf{\Lambda}$ will denote an actor mapping, and hence $\mathbf{\Lambda}(a)$ denotes the concrete actor corresponding to abstract actor $a$. We will also apply actor mappings to sets of abstract actors and to abstract locks and lock sets; the effect in each case is to replace each abstract actor with the corresponding concrete one.

One other important thing to realise is that since actors and locks have runtime representations, and can be manipulated and queried at runtime, they are subject to the same possibilities for information flows as the memory. This means that to ensure that all information flows are properly specified and tracked, locks and actors must have policies too, to govern how they may be used in a program. This is the reason for the slightly different formulation here compared to that in the previous chapter (2.3). Since locks and actors are represented at runtime, they can also carry information that influences computation, and as such they must be handled just like other parts of the state. Hence we merge the memory and lockstate components from the formulation in section 2.3 into a single state component, which also includes the actors.

For a given state component $t$ we write $pol(t)$ to denote the policy of $t$.

### 3.4.2 Validating flows

Just like for flow locks, Paralocks permit fine-grained flows where data can be effectively declassified to an actor in a series of steps, each removing one condition (i.e. lock) that needs to be fulfilled. Thus our "levels" again need to account for both actors and lock sets. We thus define an attacker $A$ to be an actor paired with a set of locks which we denote the *capability* of that attacker. The intuition is that an attacker $A = (\mathbf{a}, \mathbf{\Sigma})$ may see any data

guarded from actor $\mathbf{a}$ by at most $\mathbf{\Sigma}$. Formally,

$$A \in \mathit{Actors} \times \mathcal{P}(\mathit{Locks})$$

We write $\mathbf{Cap}(A)$ for the capability of $A$. Note though that attackers observe concrete things at runtime, so they represent concrete actors with concrete capability sets. This is different from the flow locks model, despite the overall similarity.

As we saw for flow locks, to formulate security in a "knowledge evolution" style we need a number of auxiliary definitions.

A *trace* is a sequence of labels denoting changes to the state. An *A-observable trace* is a trace where we mask out changes to pieces of the state that the attacker $A$ cannot see. We say that an attacker $(\mathbf{a}, \mathbf{\Sigma})$ *can see* some part of the state with policy $p$ iff $p \sqsubseteq \{\mathbf{\Sigma} \Rightarrow \mathbf{a}\}$. A transition is *visible to* $A$ if $A$ can see the portion of the state involved in the change. We write $\langle c, S\rangle \xrightarrow{w}_A \langle c', S'\rangle$ when $\langle c, S\rangle \xrightarrow{w} \langle c', S'\rangle$ and the transition is visible to $A$, and

$$\langle c, S\rangle \xRightarrow{\vec{w}}_A \langle c', S'\rangle$$

when there exists a sequence $\vec{\ell}$ of labeled transitions between the respective configurations, where the projection of $\vec{\ell}$ to the non-silent $A$-visible transitions is equal to $\vec{u}$. We sometimes omit result configurations if we only care about the output of a program, as in $\langle c, S\rangle \xRightarrow{\vec{w}}_A$ . Note that the series of execution steps generating a trace need not be maximal, so the set of all $A$-observable traces of a given program-state pair for a given attacker $A$ is prefix closed.

An *A-low state* is a projection of a state to exactly those parts visible to attacker $A$.

With these definitions in hand, we can define the notion of *attacker knowledge* as follows:

**Definition 17** (Attacker knowledge)**.** The knowledge an attacker has of the starting memory after observing trace $\vec{u}$ of program $c$ with a starting state who's $A$-low projection is $L$ is

$$k_A(\vec{u}, c, L) = \left\{ S \mid S \sim_A L, \langle c, S\rangle \xRightarrow{\vec{w}}_A \right\}$$

i.e., the set of all possible starting states that might lead to that trace.

Note that knowledge grows (uncertainty decreases) during execution, so we always have that $k_A(\vec{u}u, c, L) \subseteq k_A(\vec{u}, c, L)$.

### 3.4.3 Paralocks Security

To validate that all information flows in a program are secure according to the stated policies, each output must be examined in the context it takes place, which in our flow locks setting means the lock state in effect at the time of the output. Consider for example the simple program $x := y$, where $pol(x) = \{a\}$ and $pol(y) = \{\sigma \Rightarrow a\}$. Clearly this program is insecure in isolation, since the policy on $x$ is less restrictive than that on $y$, but it would be secure providing that $\sigma$ was already open.

To help with our definition, we first define the notion of an *A-observable run* of a program to be a non-empty $A$-observable trace of the program, paired with the lockstate in which the last output of that trace takes place. We formally define the set of all $A$-observable runs that could arise from a given program $c$ starting in a state whose $A$-low projection is $L$, as

**Definition 18** ($A$-observable run)**.**

$$Run_A(c, L) = \left\{ (\vec{u}u, \mathbf{LS}(S')) \mid S \sim_L, \langle c, S\rangle \stackrel{\vec{w}}{\Longrightarrow}_A \langle c', S'\rangle \stackrel{w}{\rightarrow}_A \right\}$$

Now, for a given attacker, representing a particular split of the state into high and low portions, who observes an output, the requirement is that this output may not signify a data flow from "high" to "low" portions of the state, unless the lock state permits such flows. Note that a single attacker is a very course-grained representation of security, as it is *only* able to distinguish between "high" and "low", but no nuances. As a consequence, *any* lockstate that would allow *some* flow from high to low will do. The split of high and low depends on the capability of the attacker, so for an attacker $A$ we have that some lockstate $\mathbf{\Sigma}$ allows flows from high to low as long as $\mathbf{\Sigma} \not\subseteq \mathbf{Cap}(A)$. If $\mathbf{\Sigma} \subseteq \mathbf{Cap}(A)$ then the only flows that are allowed fall completely inside the parts of the state that $A$ considers low.

The fine granularity is obtained by quantifying over all possible such attackers, since for any bad flow there must exist an attacker for which the flow is from "high" to "low", but without a permissive enough lockstate.

Our formal definition of top-level security for a program, denoted $\mathbf{PLS}(c)$, can then be defined in terms of runs as follows:

**Definition 19** (Paralock security)**.** A program is said to be *Paralocks secure*, written $\mathbf{PLS}(c)$, if for all attackers $A$, for all $A$-low states $L$, for all runs $(\vec{u}u, \mathbf{\Delta}) \in Run_A(c, L)$ we have that if $\mathbf{\Delta} \subseteq \mathbf{Cap}(A)$ then

$$k_A(\vec{u}u, c, L) = k_A(\vec{u}, c, L)$$

Informally, if the lock state at the time of the update would not allow any flows from "high" to "low" portions of the state, then no knowledge may be gained about the initial state.

In practice we also need a generalised definition which accounts for subprograms that are secure in the context they appear, where "context" here means the actors which exist and the locks which are open. This definition is slightly trickier to achieve than for simple flow locks, hence we give it some more in-depth treatment.

Consider the program **open** $\sigma(a); x := y$ where $pol(x) = \{a\}$ and $pol(y) = \{\sigma(a) \Rightarrow a\}$. This program is intuitively secure, even though the second half is not secure in isolation. For the second subprogram to be secure, it must exist in a context where the actor variable $a$ maps to some concrete actor $\mathbf{a}$, and the concrete lock $\sigma(\mathbf{a})$ is open.

To generalise our security definition, we first generalise the notion of runs to account for the fact that we may rely on some locks being open, with respect to a particular actor mapping. Formally,

**Definition 20** (Generalised $A$-observable run)**.** Let us say that state $S$ is *compatible with* $\Sigma$ if $\mathbf{LS}(S) \supseteq \mathbf{Act}(S)(\Sigma)$. Similarly we say that state $S$ is compatible with an actor set $\Lambda$ if $\mathrm{dom}(\mathbf{Act}(S)) \supseteq \Lambda$.

We then have the following definition:

$$Run_A(\Sigma, \Lambda, c, L) = \{\vec{u}u, \mathbf{LS}(S') \mid$$
$$S \sim_A L, \Lambda \text{ and } \Sigma \text{ are compatible with } S, \langle c, S\rangle \stackrel{\vec{w}}{\Longrightarrow}_A \langle c', S'\rangle \stackrel{w}{\rightarrow}_A \ \}$$

The added condition that $\Lambda$ and $\Sigma$ are compatible with $S$ states that we only consider states that have bindings for at least the abstract actors in $\Lambda$ and that we only care about states where the lock state has at least the locks in $\Sigma$ open, with respect to the actor mapping used.

This definition is clearly a generalisation of the previous top-level definition, and we get the specialised version by letting $\Lambda$ and $\Sigma$ be empty sets.

The generalised security definition now comes for free, we just need to use the generalised version of runs:

**Definition 21** (Generalised Paralocks security)**.** A program $c$ is said to be $(\Lambda, \Sigma)$ Paralocks secure, written $\mathbf{PLS}(\Lambda, \Sigma, c)$, if for all attackers $A$, for all $A$-low states $L$, for all runs $(\vec{u}u, \mathbf{\Delta}) \in Run_A(\Lambda, \Sigma, c, L)$ we have that if $\mathbf{\Delta} \subseteq \mathbf{Cap}(A)$ then

$$k_A(\vec{u}u, c, L) = k_A(\vec{u}, c, L)$$

Unsurprisingly we can do the same specialisation here to get $\mathbf{PLS}(c) = \mathbf{PLS}(\emptyset, \emptyset, c)$.

The above definition of security is termination sensitive, just like with our definition for simple flow locks. We employ the same trick as for flow locks to get a weaker but more easily verified termination insensitive version as follows:

**Definition 22** (Termination-insensitive Paralocks Security)**.** A program $c$ is said to be termination insensitive $(\Lambda, \Sigma)$ Paralocks secure, written $\mathbf{PLS_{TI}}(\Lambda, \Sigma, c)$, if for all attackers $A$, for all $A$-low states $L$, for all pairs of runs which differ only at the last output $(\vec{u}u, \mathbf{\Delta}), (\vec{u}u', \mathbf{\Delta}') \in \mathit{Run}(\Lambda, \Sigma, c, L)$ we have that if $\mathbf{\Delta} \subseteq \mathbf{Cap}(A)$ then

$$k_A(\vec{u}u, c, L) = k_A(\vec{u}u', c, L)$$

One important property of generalised Paralocks security is monotonicity.

**Theorem 2** (Monotonicity of Paralock Security)**.** *If* $\mathbf{PLS}(\Lambda, \Sigma, c)$ *and* $\Sigma' \supseteq \Sigma$ *and* $\Lambda' \supseteq \Lambda$ *then* $\mathbf{PLS}(\Lambda', \Sigma', c)$

The proof follows directly from the definition of $\mathbf{PLS}(\Lambda, \Sigma, c)$, and in particular from the fact that $\mathit{Run}(\Lambda, \Sigma, c, L)$ only considers states that are compatible with $\Lambda$ and $\Sigma$. Fewer states will be compatible with larger actor and lock sets, so the security requirement is weakened.

## 3.5 Enforcement: A Sound Paralocks Type System

In this section we give an example of how Paralocks can be combined with a concrete programming language, and present a type system which guarantees that well-typed programs are (termination insensitive) Paralocks secure. The underlying language we present is as simple as possible while still using the full expressive power of Paralocks, to focus on the interesting parts of the interaction.

**Expressions:** $e \ ::= \ n \mid x[\vec{a}] \mid e \oplus e$

**Commands:**

$c \ ::= \ x[\vec{a}] := e \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } (e)\ c \mid \textbf{skip}$
$\quad \mid c_1; c_2 \mid \textbf{open } \sigma(\vec{a}) \mid \textbf{close } \sigma(\vec{a}) \mid \textbf{newactor } a \textbf{ in } c$
$\quad \mid \textbf{when } \sigma(\vec{a}) \textbf{ do } c \textbf{ else } c \mid \textbf{forall } \sigma(\vec{a}) \textbf{ do } c$

**Internal Commands:** $c ::= \textbf{for } \sigma(\vec{a}) \textbf{ in } \mathbf{\Sigma} \textbf{ do } c$

**Figure 3.2:** Example language syntax

The language, found in Figure 3.2, is at its core a sequential imperative language, with assignments, conditionals and loops. Data sinks and sources are kept abstract and are uniformly represented as references, with each reference having an attached policy. For simplicity the only basic type is the integers. The internal command (**for**) is not part of the surface syntax and only arises in the operational semantics.

To manipulate locks we introduce the commands **open** $\sigma(\vec{a})$ and **close** $\sigma(\vec{a})$. These are the only commands in the language that can change the lock state component of the state. Unlike in the basic flow locks language, locks here will have runtime representations and can carry information, so the runtime use of locks will also be governed by policies. The **when** command is a conditional which queries the the state of a particular lock.

New actors can be introduced dynamically using the **newactor** $a$ command, which generates a fresh concrete actor and brings a new actor variable $a$ into scope for the enclosed subcomputation. Note that this could for instance be placed inside a loop, so the same variable name introduced by the same **newactor** command can represent many different concrete actors during execution.

In order to keep the language and in particular the type system simple, actors are not first class entities. To regain some of the lost expressive power from this choice, we reuse lock families as a sort of storage for actors. A lock family can be viewed as a "named collection of actors", and to access the contents of such a collection we introduce the **forall** command, which loops over all open locks in some family, bringing the relevant actors into scope in the loop body for each iteration. We assume that the order in which locks are looped over is deterministic.

The creation and use of actors may also be a conduit for information flow at runtime, so like references and locks we could require actor variables to have policies too. For simplicity though, we assume that all actors introduced by **newactor** commands are public, i.e. with a policy $\{\forall x.\, x\}$. Actor variables bound by a **forall** command will carry information about the lock family used in the loop, so we assume they inherit the policy of that lock.

Regarding policies, it is important to note that the runtime policies on runtime entities will talk about concrete actors and locks, while in the program code the policies will mention abstract entities. We have no explicit declaration of references in the language, instead we assume that they are globally available. But since actor variables are *not* globally defined, this has the effect that policies on references (and locks) cannot contain free actors, as that could lead to name capture problems. In many settings this would be too restrictive, since it would preclude actor-specific data.

To enable actor-specific data while avoiding all the extra machinery that

would have been needed to track scoping and name-capture problems for policies, we instead make this explicit at the top level by having actor-parametrised *families* of references. For full flexibility we allow any number of parameters on a family of references, just as with locks.

Locks are also globally available, and may have actor parameters. However, for simplicity we do not allow the policies on lock families to mention the actor parameters, and thus may not contain any free actor variables. In other words, for a family of references we could have different actors having access to each individual reference, e.g. $pol(x[a]) = \{a\}$, whereas for families of locks we only allow a single policy for the entire family, e.g. $pol(\sigma) = \{\forall x.\, \sigma(x) \Rightarrow x\}$.

With all this in place, there is no need for any control that actors in policies refer to the proper runtime actors, since they cannot appear free in policies.

To illustrate these language features consider a simple sealed-bid auction scenario. For example, if we wanted a 'bid' variable for each bidder in a sealed-bid auction, we could model that with a family of references $\mathsf{bid}[a]$, parametrised by actors. Policies on such families can then use the actor parameter, so we could have

$$pol(bid[a]) = \{a; \forall x.\, \{\mathsf{Bidder}(x), \mathsf{AuctionClosed}\} \Rightarrow x\}$$

where the policy on the individual references in the 'bid' family depends on the actor in question. As an example, the code representing the registration of a new bidder might be written:

```
newactor b in
  open Bidder(b)
  bid[b] := getBid
```

where we assume that `getBid` is an input channel from the actor in question, represented as a reference. The policy on the reference `bid[b]` would be $\{b; \forall x.\, \{\mathsf{Bidder}(x), \mathsf{AuctionClosed}\} \Rightarrow x\}$, stating that all bidders can gain information about this bid once the auction is completed.

The code fragment for concluding the auction and publishing the winning bid (the first of the largest bids) could then be written:

```
maxBid := 0
forall Bidder(x) do
  if bid[x] >= maxBid then
    maxBid := bid[x]
    forall Winner(y) do close Winner(y)
    open Winner(x)
  else skip
open AuctionClosed
```

To be able to compute the maximum bid before the auction is marked as closed (as in this example) we would give `maxBid` the policy

$$\{\forall x.\, \{\mathsf{Bidder}(x), \mathsf{AuctionClosed}\} \Rightarrow x\}$$

We use a separate lock family to denote the winning actor, and by (line 5) closing all previous winners and then opening the lock for the new winner, we are assured that we only ever have (at most) one winner. We could then loop over all actors $a$ for whom the $\mathsf{Winner}(a)$ lock is open, to do specific things relating to the winner.

Again we stress that while some of the language design choices here are unorthodox, this is just a consequence of keeping the language and type system relatively small. In a more realistic programming language incorporating Paralocks, there are a number of other language design considerations. Supporting first-class dynamic actors would be a more natural route in a richer language, and this would be naturally supported in the type system using singleton types. From the expressiveness viewpoint support for policies not known until runtime (cf. DLM runtime labels [ZM07a]) could well prove useful, but would require more language features to enable static checking. However, the issues involved there are largely problems of *enforcement*. While interesting in their own right, they are orthogonal to the core issues of this work, namely the Paralocks policy specification language and its associated definition of security.

We discuss all these issues in chapter 4.

### 3.5.1 Operational Semantics

The operational semantics of our example language can be found in Figure 3.3. Transitions occur between configurations of the form $\langle c, S\rangle$, where $c$ is the command and $S$ is the program state. This state is a triplet of an actor mapping ($\mathbf{Act}(S)$), a lock state ($\mathbf{LS}(S)$) and a memory ($\mathbf{Mem}(S)$).

$$\langle n, S\rangle \Downarrow n \qquad \frac{\langle e_1, S\rangle \Downarrow v_1 \quad \langle e_2, S\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, S\rangle \Downarrow v_1 \oplus v_2}$$

$$\left.\begin{array}{c} \langle x[\vec{a}], S\rangle \Downarrow \mathbf{Mem}(S)[x[\vec{\mathbf{a}}]] \\ \langle \mathbf{open}\ \sigma(\vec{a}), S\rangle \xrightarrow{\mathbf{open}\ \sigma(\vec{\mathbf{a}})} \langle \mathbf{skip}, S \cup \{\sigma(\vec{\mathbf{a}})\}\rangle \\ \langle \mathbf{close}\ \sigma(\vec{a}), S\rangle \xrightarrow{\mathbf{close}\ \sigma(\vec{\mathbf{a}})} \langle \mathbf{skip}, S \setminus \{\sigma(\vec{\mathbf{a}})\}\rangle \\ \dfrac{\langle e, S\rangle \Downarrow v}{\langle x[\vec{a}] := e, S\rangle \xrightarrow{x(v)} \langle \mathbf{skip}, S[x[\vec{\mathbf{a}}] \mapsto v]\rangle} \end{array}\right\} \vec{\mathbf{a}} = \mathbf{Act}(S)(\vec{a})$$

$$\frac{\langle e, S\rangle \Downarrow v \qquad v \in \{\mathbf{true}, \mathbf{false}\}}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_{\mathbf{true}}\ \mathbf{else}\ c_{\mathbf{false}}, S\rangle \xrightarrow{\tau} \langle c_v, S\rangle}$$

$$\langle \mathbf{while}\ (e)\ c, S\rangle \xrightarrow{\tau} \langle \mathbf{if}\ e\ \mathbf{then}\ (c; \mathbf{while}\ (e)\ c)\ \mathbf{else}\ \mathbf{skip}, S\rangle$$

$$\frac{\langle c_1, S\rangle \xrightarrow{\ell} \langle c_1', S'\rangle}{\langle c_1; c_2, S\rangle \xrightarrow{\ell} \langle c_1'; c_2, S'\rangle} \qquad \langle \mathbf{skip}; c_2, S\rangle \xrightarrow{\tau} \langle c_2, S\rangle$$

$$\langle \mathbf{newactor}\ a\ \mathbf{in}\ c, S\rangle \xrightarrow{a(\mathbf{a})} \langle c, S[a \mapsto \mathbf{a}]\rangle \qquad (\mathbf{a}\ \text{fresh})$$

$$\langle \mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2, S\rangle \xrightarrow{\tau} \begin{cases} \langle c_1, S\rangle & \sigma(\mathbf{Act}(\vec{a})) \in \mathbf{LS}(S) \\ \langle c_2, S\rangle & \text{otherwise} \end{cases}$$

$$\frac{\mathbf{\Sigma} = \{\vec{\mathbf{a}} \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(S)\}}{\langle \mathbf{forall}\ \sigma(\vec{a})\ \mathbf{do}\ c, S\rangle \xrightarrow{\tau} \langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c, S\rangle}$$

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \{\vec{\mathbf{a}}\} \cup \mathbf{\Sigma}\ \mathbf{do}\ c, S\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})} \langle c; \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c, S'\rangle$$
$$\vec{a} = a_1, \ldots, a_n \qquad S' = S[a_1 \mapsto \mathbf{a}_1, ..., a_n \mapsto \mathbf{a}_n]$$

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \emptyset\ \mathbf{do}\ c, S\rangle \xrightarrow{\tau} \langle \mathbf{skip}, S\rangle$$

**Figure 3.3:** Operational Semantics

For simplicity we lift updates on individual components to the full state, so for instance we write e.g. $S[x \mapsto v]$ to update the value of a variable in the memory, or $S \cup \sigma$ to add an open lock to the lock state. Since the three components have disjoint domains there should be no risk for confusion.

Apart from the labels on transitions, there should be no surprises in the rules for the ordinary imperative constructs. Regarding **open** and **close**, the only thing of note is that we need to map actor variables in locks to their concrete representations before updating the lock state.

The **when** command is very similar to the standard **if**, the only difference being that **when** queries the lock state instead of the memory.

The **newactor** command generates a fresh concrete actor representation and binds it to the variable name given. Since we assume all actors bound this way are public, we don't need to care about the particulars of the generation scheme. Syntactically the variable is scoped, but in the semantics we don't bother to remove it once we leave the scope, instead we rely on the type system to ensure that there can be no accidental capture.

Finally, the most complex command semantically is the **forall**, which loops over all locks in some particular family. Its execution is done in two steps. First, the set of locks in the family that are open is (deterministically) calculated, and second that set is looped over, one lock at a time. For this we need to extend the language with an internal command **for** $\sigma(a_1, ..., a_n)$ **in** $\mathbf{\Sigma}$ **do** $c$, to handle the actual looping. The transition rule for **forall** is then simple: gather all open locks in the relevant lock family and go to the next step, the **for**.

In the **for** we bind the relevant actors to the provided variables and then proceed to execute the body. Just like with the **newactor** rule, we don't care where the scope of the variables ends syntactically, relying on the type system to handle the scoping details.

The transition arrows are labeled with outputs that signal all direct information flows that take place during execution, which in this simple language means all changes to the program state. These are purely for the sake of reasoning about security and otherwise have no effect on the computation. The commands that have an effect on the state are assignments for the memory and **open** and **close** for the lock state. For the actor mapping, the **newactor** command can introduce a single new actor in scope, while the **forall** loop, via the auxiliary internal **for** construct, can bind a number of names in one transition step. All other base rules have no effect on the state, and thus yield the silent output $\tau$.

### 3.5.2 Type System

To enforce security we use the type system in Figure 3.4. Since we only have integers as the base type for values, we don't need to track base types at all, so our type system only handles the security component.

For expressions, the typing judgment is simply $\Lambda \vdash e : r$, where $r$ is a Paralocks policy which we call the *read effect* of the expression, as it intuitively specifies who may read data from references with this policy. In effect it will be the least upper bound of the policies on references used to compute the expression $e$. There should be no surprises in how this read effect is computed, though note that the rule for references handles both parametrised and unparametrised references, as we allow the vector of actors to be of length 0.

The typing judgment for commands is a bit more involved, but the various components should come as no surprise. The judgment is

$$\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'$$

where $c$ is the command to type and $w$ is a policy we call the *write effect* of the command. Intuitively this policy specifies who would be able to notice that the command was executed, by observing its effects on the state. It is thus the greatest lower bound of all policies on references, locks and actor variables whose values are affected by the command. The purpose of this policy is to track indirect flows, similar to the use of a "program counter" in many other systems. This can be seen in the rules for the commands that affect control flow, namely **if**, **while**, **when** and **forall**. All these rules compare the policy of the branching expression or lock with the write effect of the body of the command.

The write effect is straightforward to compute for most rules. For assignments, **open** and **close** it is simply the policy of the affected location or lock. The **newactor** command introduces actor variables with the policy $\bot$, which is thus its write effect, as $\bot$ is clearly at least as liberal as any write effect of the body. The most interesting rule in this regard is that of the **forall** command. We cannot in general know exactly which actors will be referenced in the loop iterations, so we assume it may be any of them, meaning that actors introduced by the **forall** that appear in the write effect of the body must be universally quantified. However, since the **forall** also binds actors to the relevant variables, and these variables inherit the policy of the lock, the write effect of the whole command will be exactly the policy of the lock, since we require that to be more liberal than any write effect of the body.

$$\frac{}{\Lambda \vdash n : \bot} \qquad \frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])}{\Lambda \vdash x[\vec{a}] : pol(x[\vec{a}])} \qquad \frac{\Lambda \vdash e_1 : r_1 \quad \Lambda \vdash e_2 : r_2}{\Lambda \vdash e_1 \oplus e_2 : r_1 \sqcup r_2}$$

$$\frac{\vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash \textbf{open}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

$$\frac{\vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash \textbf{close}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \simeq \vec{b}\}}$$

$$\frac{}{\Lambda; \Sigma \vdash \textbf{skip} \rightsquigarrow \top, \Sigma} \qquad \frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq pol(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow pol(x[\vec{a}]), \Sigma}$$

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma \vdash \textbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

$$\frac{\Lambda; \Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Lambda; \Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

$$\frac{\begin{array}{c}\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \\ pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)\end{array}}{\Lambda; \Sigma \vdash \textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \vec{a} \cap \Lambda = \emptyset}{\Lambda; \Sigma \vdash \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash \textbf{newactor}\ a\ \textbf{in}\ c \rightsquigarrow \bot, \Sigma' \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}}$$

$$\frac{\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash c} \quad \text{(Top level judgement)}$$

**Figure 3.4:** Paralocks Type System

$\Lambda$ is the set of actors in scope, for both commands and expressions, and the **newactor** and **forall** commands introduce new actors into this scope as expected. We use it only to ensure that any mention of actor variables as arguments to references and locks are done in a correct way, and that no variable names clash.

$\Sigma$ is the set of locks assumed to be open when the command starts executing, and $\Sigma'$ is a lower bound on the locks that will be open afterwards. The one place where $\Sigma$ is actually used is in the assignment rule. In this rule we must determine whether the policy of the expression is compatible with the policy of the variable *relative to the current lock state*. The idea is the same as with flow locks – but the details are more complex. For example, suppose we have a policy $p = \{a, \forall x.\, \mathit{Actsfor}(a, x) \Rightarrow x\}$. Intuitively this says that $a$ may always read the data, and that for any actor $x$, if the lock $\mathit{Actsfor}(a, x)$ ("flow from $a$ to $x$ is permitted") then $x$ may also read. If we specialise this policy to a lock state $\Sigma = \{\mathit{Actsfor}(a, b)\}$ then the policy in force at that state $p(\Sigma)$ is $\{a, \forall x.\, \mathit{Actsfor}(a, x) \Rightarrow x, b\}$. I.e. in that state, $b$ is also unconditionally permitted to see the data. Specialisation is most easily understood in logical terms: $p(\Sigma)$ is just the most liberal policy which is entailed by the conjunction of $p$ and $\Sigma$.

In the definitions that follow we distinguish $\forall$-bound actor variables syntactically, using metavariable $x$.

**Definition 23** (Matching)**.** Let $\theta$ be a substitution from bound actor variables to free actor variables. We say that $\Sigma$ *matches* $\Sigma'$ *with* $\theta$ if and only if the set of bound actors in $\Sigma$ is equal to the domain of $\theta$, and $\Sigma\theta \equiv \Sigma'$.

For example, $\{\mathit{Actsfor}(a, x)\}$ matches $\{\mathit{Actsfor}(a, b)\}$ with $[b/x]$.

**Definition 24** (Specialisation)**.** For a policy $p$ and a lock state $\Sigma$, we define the normalisation of $p$ at $\Sigma$, written $p(\Sigma)$, as

$$p(\Sigma) = \bigcup_{c \in p} \{c \cdot \Sigma\}, \qquad \text{where}$$

$$(\forall \vec{x}.\, \Delta \Rightarrow b) \cdot \Sigma \stackrel{\text{def}}{=} \{\forall \vec{x}.\, \Delta_2\theta \Rightarrow b\theta \;\mid\; \Delta \equiv \Delta_1 \cup \Delta_2;\ \Sigma_1 \subseteq \Sigma;\ \Delta_1 \text{ matches } \Sigma_1 \text{ with } \theta \;\}$$

Note that $p(\Sigma)$ always contains $p$ (to see this take $\Delta_1$ and $\Sigma_1$ to be the empty set in the auxiliary definition above) – i.e. $p(\Sigma) \sqsubseteq p$ – normalising a policy always yields a more liberal policy.

Computing the outgoing lock state is straightforward in most cases, but a few rules are slightly complex. Actors introduced by **newactor** and **forall** are scoped, and when their respective scopes end we need to forget about any

locks mentioning those actors, to avoid name clashes with potential future scopes reusing the same actor variable.

Most interesting perhaps is the rule for **close**, which has to account for potential aliasing issues between actor variables. Hence it is maximally pessimistic, and assumes that not only the lock that is explicitly mentioned will be closed, but also any other lock in the same family where the actor arguments may point to the same concrete actors at runtime. Two variables introduced by **newactor** commands can never be aliases of each other as they must represent fresh concrete actors. A variable introduced by a **forall** could alias any other variable though. We assume an implicit predicate *alias* where $alias(a) = \textbf{true}$ if $a$ in the current scope is introduced by a **forall** construct, otherwise **false**. The result clearly depends on the context in which the function is called. We then define a simple may-alias relation as

$$a \simeq b \stackrel{\text{def}}{=} alias(a) \vee alias(b) \vee a = b.$$

We extend this relation to equal-length vectors of actors in a point-wise manner. Using this may-alias relation, the rule for **close** is suitably pessimistic about what abstract locks may actually be closed at runtime.

### 3.5.3 Security

We show that well typed programs are Paralocks secure. The proof can be found in the appendix. Here we just note the main technical stepping stones – the first of which is the standard property that reduction preserves typability:

**Lemma 2** (Preservation)**.** *Let us say that state $S$ is* compatible with $\Sigma$ *if* $\mathbf{LS}(S) \supseteq \mathbf{Act}(S)(\Sigma)$*. Similarly we say that state $S$ is compatible with an actor set $\Lambda$ if* $\mathrm{dom}(\mathbf{Act}(S)) \supseteq \Lambda$*.*

*Now suppose that $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ and $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$. Then if $\Lambda$ and $\Sigma$ are compatible with $S$ then $\Lambda'; \Sigma' \vdash c' \rightsquigarrow w', \Delta'$ for some $\Lambda'$ and $\Sigma'$ compatible with $S'$, $w \sqsubseteq w'$ and $\Delta \subseteq \Delta'$.*

The second basic property is the global ("big step") property of the effect components of the typing derivation. Stated informally (to convey the intuition without all the technicalities), they say that whenever $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ then

- If data labeled $w$ is not visible to attacker $A$ then any computation of $c$ in any start state compatible with $\Sigma$ will not produce any $A$-visible output (and hence will not modify the parts of the state with policy $w$ or stronger).

- A terminating computation of $c$ in a state with at least locks $\Sigma$ open will result in at least locks $\Delta$ being open.

Finally we have proven the main theorem of this section, namely that a well-typed program is guaranteed to be secure by our semantics for Paralocks. Since the type system as stated is termination insensitive, for instance it allows "high loops" to precede "low writes", we formally have that well-typed programs are termination-insensitive Paralocks secure:

**Theorem 3.** *If* $\emptyset;\emptyset \vdash c$ *then* $c$ *is termination-insensitive Paralocks secure (*$\mathbf{PLS_{TI}}(c)$*).*

This in turn is a corollary of a theorem involving a generalisation of the $\mathbf{PLS_{TI}}$ property. Again, the details are available in the appendix.

## 3.6 Example Encodings

As we showed in the previous chapter, flow locks can be used to encode many different declassification ideas. Since Paralocks is an extension, it can obviously encode all the same ideas, but can unsurprisingly also express and encode some further idioms that with just flow locks would lead to prohibitively cumbersome encodings.

### 3.6.1 Robust Declassification

Our encoding of robust declassification in the previous chapter (section 2.6.3) had to resort to a trick with an auxiliary log file to enable robustness of the declassify operation. With Paralocks, where locks are also given policies, we can do away with the log file and just put the requisite policy $\{\mathit{trusted}\}$ directly on the respective $\sigma_i$ locks that control confidentiality. These locks then cannot be opened or closed in untrusted contexts.

### 3.6.2 The Decentralised Label Model

To show the true flexibility of Paralocks, we show how it can be used to encode the Decentralised Label Model (DLM) of Myers and Liskov [ML97]. In the previous chapter we hinted at a possible DLM encoding using flow locks, but that encoding would require all principals to be known statically, so that all relations between principals could be "hard wired" into the policy. Paralocks now lets us give a much more convenient encoding.

The core component of the DLM is the *label*. Data is decorated with labels that govern how that data may be used. A label $L$ specifies the *owners* of some data, written $owners(L)$, and for each owner the set of *readers* allowed by that owner, $readers(L, o)$. The intuition behind the owners is that data, at its origins, has a single owner who specifies its readers. The label on a piece of data reflects the various potential origins of the information in that data.

The decentralisation relates to the readers. Each owner can independently specify who they consider trusted to view the data. The *effective readers* of some data are those for whom all the owners have agreed may read it, i.e. the intersection of the separate reader sets for all owners. A label for some data may look like

$$\{o_1 : r_1, r_2 \ ; \ o_2 : r_2, r_3\}$$

where $o_1$ and $o_2$ are owners and $r_1, r_2$ and $r_3$ are readers. Such data might be obtained by combining data from $o_1$ and $o_2$ in some way. The effective readers in this example is just $r_2$.

A label $L_2$ is said to be more restrictive than label $L_1$, written $L_1 \sqsubseteq_{DLM} L_2$, if it has at least the same owners, and each of those owners list fewer potential readers. Formally it is defined [ML97] as

$$L_1 \sqsubseteq_{DLM} L_2 = owners(L_1) \subseteq owners(L_2) \wedge \\ \forall o \in owners(L_1).\, readers(L_1, o) \supseteq readers(L_2, o)$$

Data may be *relabeled* in two ways, through an assignment:

- Data with label $L_1$ can always be assigned to a storage location (a container) with label $L_2$ if $L_2$ is more restrictive than $L_1$, i.e. $L_1 \sqsubseteq L_2$.
- Data can be *declassified* by adding more readers for a given owner. In the DLM this can be done freely providing that the current process runs on behalf of the owner in question.

Apart from labels, there is one other important component to the DLM, namely the *principal hierarchy* and its associated acts-for relationship. The DLM lets principals represent both individual users and other notions like roles and groups, and membership for a user in a role can thus be modeled by letting the user act for that role.

The acts-for hierarchy has two effects on the security of a program. First, if $a$ acts for $b$ and $b$ is listed as a reader in a label, then $a$ is also implicitly a reader. Second, if a piece of code runs on behalf of $a$, then it also implicitly runs on behalf of $b$, so code running on behalf of $a$ may conduct declassifications in $b$'s name.

To encode the DLM using Paralocks, we need to represent a number of things explicitly that are implicit in the DLM. The first of these is the acts-for relationship that we just discussed. If the principal hierarchy states that $b$ acts for $a$, then we expect the $\mathit{ActsFor}(a, b)$ lock to be open. We can account for changes to the hierarchy during execution by opening or closing the appropriate locks.

The acts-for relationship is transitive and reflexive. Each actor acts for himself, and if $a$ acts for $b$ and $b$ acts for $c$, then we assume implicitly that $a$ acts for $c$ as well. With Paralocks, properties like transitivity and reflexivity are not built in. Locks are just boolean variables with no additional predefined semantics attached to them. If we want a transitive property for a particular relation like *ActsFor*, we must handle this explicitly.

A naive attempt could be to try to handle this on the policy level, e.g. by specifying the policy as

$$\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x; \forall x, y.\, \mathit{ActsFor}(a, x), \mathit{ActsFor}(x, y) \Rightarrow y\}$$

This is not a viable approach since the above policy only works for one step of transitivity; for full transitivity we would need to explicitly list the transitive closure, and this would be at best cumbersome, and impossible if we could not statically enumerate all actors. There are two routes to deal with this issue. The first is to extend the expressive power of the policy language to enable such global invariants to be expressed as part of the policy, and reflected in the security model. This is explored in section 3.7. For now we take a simpler route, and view transitivity not as a property of a policy, but rather an intended invariant on the set of locks open at any given time. This invariant can easily be maintained at runtime by suitably encapsulating lock manipulation operations.

So if we ensure that any program using a policy involving delegation maintains the transitivity property of *ActsFor*, then it is enough for the policy to be stated (as before) as simply

$$\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x\}.$$

Second, to account for declassification being possible only when the process runs with the authority of the owner of the declassified data, we need a lock family $\mathit{RunsFor}(a)$. We expect the appropriate locks to be open for those actors for whom the code runs. Further, we also expect the invariant that whenever $\mathit{ActsFor}(a, b)$ and $\mathit{RunsFor}(b)$ is open then $\mathit{RunsFor}(a)$ is also open, again making the implicit relationship explicit.

Third, since Paralocks take the perspective of the reader, as opposed to the owner as in in DLM, the policy needs to be explicit about the potential

future readers to whom the data may be declassified. With respect to a given owner, we can freely add new readers as long as the code executes with that owner's authority. We can thus model the label $\{o :\}$, i.e. data owned by $o$ with no added readers, with the policy

$$\{\forall x.\, \mathit{RunsFor}(o) \Rightarrow x\}$$

The intuition here is that in code running with $o$'s authority, this data may be declassified to any actor $x$. [1] Adding a reader to the above policy, we get $\{o : r\}$, which we would represent as

$$\{\forall y.\, \mathit{ActsFor}(r, y) \Rightarrow y; \forall x.\, \mathit{RunsFor}(o) \Rightarrow x\}$$

The first clause here corresponds to the reader $r$. By reflexivity we will always have $\mathit{ActsFor}(r, r)$ open, and hence $r$, and anyone else who acts for $r$, will be able to read data labeled with this policy.

To handle the general case of the encoding we need to deal with the case of a *potential reader* (a reader who is a reader for one but not all owners). For these readers we need to consider the owners who do *not* permit $r$ to read the data.

**Definition 25** (Label Encoding)**.** Suppose that $r$ is a (potential or effective) reader for some label $L$, and $O$ is a subset of owners for $L$. We say that the pair $(O, r)$ is a *conflict pair* for label $L$ if

$$O = \{o \mid o \in \mathit{owners}(L), r \notin \mathit{readers}(L, o)\}\,.$$

Intuitively, $O$ are the owners who have not permitted $r$ to read data labeled $L$.

Now we can define the general encoding of Labels as policies $[\![\cdot]\!] : \mathit{Label} \to \mathit{Policy}$ by

$$\begin{aligned}[\![L]\!] = \{&\forall x.\, \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L)\} \Rightarrow x\} \\ &\cup \{\forall y.\, \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \\ &\qquad\qquad \mid (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L\}\end{aligned}$$

[1] Note that in a programming language enforcing a DLM (such as JFlow/Jif [Mye99, MZZ+06]) one might want to additionally constrain that declassification occurs at explicitly declared places in the code. This is easily modeled using regular flow locks by associating a *Declassify* lock with the portion of code which is designated as a declassification. This, however, is not part of the DLM model. We explore this further in chapter 4.

The first clause in the definition of $[\![L]\!]$ says that data can be declassified to anyone providing it is in a context which runs with the authority of all owners. Otherwise a potential reader $r$ (or anyone who acts for $r$) may read providing it does so in a context which runs with the authority of those owners who did not grant explicit access to $r$.

As an example, consider the encoding of the empty label:

$$[\![\{\ \}]\!] = \{\forall x.\, \{\} \Rightarrow x\} \cup \{\} = \{\forall x.\, x\}$$

The empty label has no owners, so implicitly anyone can read data with that label – as expressed explicitly in the flow locks encoding.

When combining labels from different data sources, the DLM simply performs the union of the respective owner policies, leaving the effective reader set implicit as the intersection of all readers. In our encoding the difference between effective and potential readers is rendered more explicit. Consider combining the two policies representing $\{o_1 : r_1, r_2\}$ and $\{o_2 : r_2, r_3\}$, which are

$$\begin{aligned}
&[\![\{o_1 : r_1, r_2\}]\!] = \\
&\quad \{\forall x.\, \mathit{RunsFor}(o_1) \Rightarrow x; \\
&\quad\ \forall y.\, \mathit{ActsFor}(r_1, y) \Rightarrow y; \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y\} \\
&[\![\{o_2 : r_2, r_3\}]\!] = \\
&\quad \{\forall x.\, \mathit{RunsFor}(o_2) \Rightarrow x \\
&\quad\ \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y; \forall y.\, \mathit{ActsFor}(r_3, y) \Rightarrow y\}
\end{aligned}$$

we get $[\![\{o_1 : r_1, r_2\} \sqcup_{DLM} \{o_2 : r_2, r_3\}]\!]$

$$\begin{aligned}
&= [\![\{o_1 : r_1, r_2\ ;\ o_2 : r_2, r_3\}]\!] \\
&= \{\forall x.\, \mathit{RunsFor}(o_1), \mathit{RunsFor}(o_2) \Rightarrow x; \\
&\quad\ \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y; \\
&\quad\ \forall y.\, \mathit{RunsFor}(o_2), \mathit{ActsFor}(r_1, y) \Rightarrow y; \\
&\quad\ \forall y.\, \mathit{RunsFor}(o_1), \mathit{ActsFor}(r_3, y) \Rightarrow y\} \\
&= [\![\{o_1 : r_1, r_2\}]\!] \sqcup [\![\{o_2 : r_2, r_3\}]\!]
\end{aligned}$$

Finally we can show that the lattice of labels in the DLM is a sublattice of the Paralocks policy lattice:

**Theorem 4.** $L_1 \sqsubseteq_{DLM} L_2$ if and only if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$. Further, $[\![L_1 \sqcup_{DLM} L_2]\!] = [\![L_1]\!] \sqcup [\![L_2]\!]$, and similarly for $\sqcap$.

The proof of these properties is given in the appendix. The relationship between $\sqsubseteq$ and $\sqsubseteq_{DLM}$ amounts to saying that the encoding is sound and complete with respect to the DLM rule for relabeling data.

What we have given here is an encoding of the DLM policy specification language only. One might expect to see a deeper comparison, in which we also compare the impact of the two on the security of programs, i.e. the formal semantic security definitions. The problem is that the DLM, or more accurately its implementation in JIF, does not have a formal semantic security model. There exist models for subsets or restricted scenarios for DLM, but it has never been covered in full. But with our encoding here, we are actually able to do just that, to provide a semantic security model for programs that use the DLM for their information flow control. Our full semantic model will be presented in the next section.

## 3.7 Recursive Paralocks

In section 3.6.2 we presented an encoding of the DLM. One aspect of a DLM policy was the treatment of the *ActsFor* relation; implicitly we required that whenever we open a lock $\mathit{ActsFor}(a, b)$ then we must also open all transitive consequences. It is intended that this invariant is implemented explicitly by encapsulating the open operation appropriately within a program which uses a DLM policy.

In this section we explore an extension to the policy language which allows us to specify such properties explicitly, avoiding the need to encode them explicitly in the program. The extension is a natural logical one: allow relations between locks and flows to be specified recursively as part of a global policy component.

In this section we briefly explore the implications of this extension to the questions of *policy* (c.f. §3.7.2), *expressiveness* (c.f. §3.6.2), *semantics* (c.f. §3.4), and *enforcement* (c.f. §3.5).

### 3.7.1 Policy

Policies will now consist of two parts. Firstly we have policies on memory objects just as before: collections of clauses which have an actor variable (bound of free) as their head. For the purposes of this section it will be useful to write a clause $\forall a_1, \ldots, a_n. \Sigma \Rightarrow a$ as $\forall a_1, \ldots, a_n. \Sigma \Rightarrow \mathit{Flow}(a)$, thus making the "may flow to"-predicate explicit. The extension we make is to add a *global policy* $G$ which is also a set of clauses. These clauses differ in that their heads may be locks – and thus they may be recursive. For example,

in a DLM encoding we would include the following two clauses in the global policy:

$$\forall y.\, \mathit{ActsFor}(y, y); \forall x, y, z.\, \mathit{ActsFor}(x, y), \mathit{ActsFor}(y, z) \Rightarrow \mathit{ActsFor}(x, z)$$

This style of policy specification is already familiar in a security context: it amounts to the use of DATALOG programs as policy specifications, and has been used in numerous logics for access control policies – e.g., [Jim01, DeT02, LMW02]. We permit one further useful feature: global policies, in addition to using locks, may also use the distinguished *Flow* predicate in their specification (see Section 3.7.2 for examples).

**Policy comparison** To compare policies $p$ and $q$ we must now take into account the global policy. We write $p \sqsubseteq_G q$ to mean that policy $q$ is more restrictive than policy $p$ in the context of global policy $G$. We can define this relation by giving a straightforward interpretation in first-order logic. As before we can interpret each clause in $p$, $q$ and $G$ as first-order Horn clauses, and sets of clauses are interpreted as logical conjunction. Then we define

$$p \sqsubseteq_G q \stackrel{\text{def}}{=} G \wedge p \models q$$

To see that this does "the right thing", consider some lock state $\Sigma$. Suppose that $\Sigma \wedge G \wedge q \models \mathit{Flow}(a)$ – i.e. that in some concrete lock state the policy $q$ permits information to flow to $a$. Then it can be readily seen that $p \sqsubseteq_G q$ ensures that $\Sigma \wedge G \wedge p \models \mathit{Flow}(a)$ – i.e., $p$ allows any flow that $q$ does.

### 3.7.2 Expressiveness

Here we consider a couple of simple examples using recursive Paralocks.

**Denning Lattices, Reloaded** The flow locks encoding of standard Denning-style information flow lattices involves identifying security levels with actors, and representing a security level $j$ by the (lock-free) policy $\{\mathit{Flow}(k) \mid j \leq k\}$. Recursive Paralocks provide several alternative ways to specify this. One example is to represent the policy for data of level $k$ as just $\{\mathit{Flow}(k)\}$. The global policy then must define the covering relation of the lattice (represented as a binary lock $\prec$), together with the rule

$$\forall x, y.\, \mathit{Flow}(x), x \prec y \Rightarrow \mathit{Flow}(y).$$

So, for example, the three point lattice $L \leq M \leq H$ would be represented by the global policy

$$\{L \prec M; M \prec H; \forall x, y.\, \mathit{Flow}(x), x \prec y \Rightarrow \mathit{Flow}(y)\}.$$

**The "Complete" DLM** In the case of the DLM encoding we already mentioned the ability to express reflexivity and transitivity for the *ActsFor* hierarchy. Similarly the invariant on the *RunsFor* lock can then be specified as

$$\forall x, y.\, \mathit{ActsFor}(x, y), \mathit{RunsFor}(x) \Rightarrow \mathit{RunsFor}(y)$$

We can also move part of each policy into the global part by applying the *ActsFor* hierarchy to *Flow* predicates:

$$\forall x, y.\, \mathit{ActsFor}(x, y), \mathit{Flow}(x) \Rightarrow \mathit{Flow}(y)$$

which says that whenever flow to $x$ is permitted then flow to $y$ is also permitted providing $y$ acts for $x$. With this rule we no longer need to be explicit about the *ActsFor* relationship in the data policy itself, so for example we can encode a label $\{o_1 : r_1; o_2 : r_1, r_2\}$ more succinctly as

$$\{\forall x.\, \mathit{RunsFor}(o_1), \mathit{RunsFor}(o_2) \Rightarrow x; r_1; \mathit{RunsFor}(o_1) \Rightarrow r_2\}$$

As an interesting side note, the original version of the DLM ([ML97]) was considered incomplete; a follow-up paper ([ML98]) identified a "complete" policy ordering. Specifically the new formulation made the policy ordering more liberal by weakening it to allow two new $\sqsubseteq$-monotone operations on policies:

- An owner $o_1$ may to be replaced in a label with an owner $o_2$ that acts for $o_1$,
- If a reader $r_1$ is listed by some owner, and $r_2$ acts for $r_1$, then we can also add $r_2$ as a reader for that owner.

Our original encoding is faithful to the original DLM, and cannot handle these components, specifically because the policy ordering was ignorant of the transitive nature of the *ActsFor* relationship and its relation to the *RunsFor* property. With the extension presented here, policy ordering becomes "complete" and thus corresponds to the revised version of the DLM [ML98].

### 3.7.3 Semantics

The definition of security needs only minor modifications to handle recursive Paralocks. There are just two places where the definition needs to be modified:

- Generalise $\sqsubseteq$ to $\sqsubseteq_G$ in the definition of the parts of a state that the attacker can see.

- In the definition of security, generalise the comparison between lock state $\mathbf{\Delta}$ and attacker capability $\mathbf{Cap}(A)$ to take into account the global policy $G$ by replacing the condition

$$\mathbf{\Delta} \subseteq \mathbf{Cap}(A) \qquad \text{(logically: } \mathbf{Cap}(A) \models \mathbf{\Delta})$$

  by $G \wedge \mathbf{Cap}(A) \models \mathbf{\Delta}$.

### 3.7.4 Enforcement

Here we consider the impact that recursive Paralocks have on the integration with the language and type system of Section 3.5.

The global policy is essentially DATALOG[2]. DATALOG and modest extensions thereof, has proved to be a popular basis for e.g. access control logics because, among other things, a query (the access control mechanism itself) can be answered in polynomial time. This is also useful in the present context. At runtime we need to enumerate all actors in a given role (the *forall*-construct in our example language), and check whether a particular lock is open (the *when* construct). It is necessary that these can be answered precisely and efficiently, and this is possible because they are DATALOG queries.

However, type checking is another matter. We do not need to answer DATALOG queries within the type system, we need to implement policy comparison ($\sqsubseteq_G$). In all other regards we conjecture that the type system given in Section 3.5 is sound for recursive Paralocks – providing we generalise the policy ordering and least-upper-bound operation accordingly.

In the case of assignment $x := e$, for example, where $x$ has policy $q$, and $e$ has policy $r$, and we know that at least locks $\Sigma$ are open, then we need to determine whether $\Sigma \wedge G \wedge r \models q$. This problem (and the similar problem of determining whether $r \sqsubseteq_G q$) is the problem of *containment* of a non-recursive DATALOG program $q$ in a recursive one $\Sigma \cup G \cup r$. This containment problem is known to be decidable, although EXPTIME-complete (see e.g. Chaudhuri and Vardi [CV97]). Whether this complexity is a problem in practice remains to be seen.

Similarly we need a generalised form of least upper bound operation where $p \sqcup_G q$ denotes the least policy $r$ such that $p \wedge G \models r$ and $q \wedge G \models r$. This kind of operation – i.e., finding a DATALOG program which gives a best

[2]Certain policies that we have used are not *safe* in the DATALOG sense (see e.g. Ceri et al [CGT89]). For example $\forall x.\, \mathit{Runsfor}(o) \Rightarrow \mathit{Flow}(x)$ is *unsafe* because $x$ does not appear in the body of the clause, and so it generates infinitely many instances. However, at any point during run time, the domains of actors is finite and known. Hence the rule can be thought of as a shorthand for $\forall x.\, \mathit{Actor}(x), \mathit{Runsfor}(o) \Rightarrow \mathit{Flow}(x)$.

approximation to the disjunction of two DATALOG programs – does not to our knowledge seem well studied in the DATALOG literature. However, we note that using $p \sqcup q$ just as before (thus ignoring $G$) would provide a safe upper bound operation which would be adequate for use in the type system.

# Chapter 4

# Paragon

## 4.1 Introduction

Paralocks is a general and powerful language for specifying information flow policies – but in itself, it is not a programming language. As noted, the intended use of Paralocks is instead as an integrated component of a programming language, either one written from scratch with Paralocks as a core feature, or as an extension to an existing language. Questions and challenges which must be addressed in the adaptation of these ideas to a real language include:

- whether the model can be scaled to handle the features of a real programming language, such as objects, exceptions, dynamic allocation, aliasing, and so forth,
- since the encoding of complex policies requires *computation* of policies (as in the DLM example from 3.6.2), how can static type-checking be used?
- Paralocks provides a "core calculus" for building information flow policies; what abstraction facilities are needed to make programming with Paralocks palatable?

In this chapter we detail how to add Paralocks as an extension to the general purpose programming language Java. The resulting language we dub Paragon – "the way programs *should* be written".

### 4.1.1 Why Java?

As shown in the previous chapter, Paralocks is agnostic in what base language it is used in conjunction with. For the task of building a full-fledged, security-typed programming language incorporating Paralocks, we have chosen to work with Java, for several reasons.

**Adoption** First of all, Java is a well-known and widely adopted language, both commercially and academically, and thus requires no further introduction. Being well spread, the chances for Paragon to be accepted and adopted clearly increase.

**Clean semantics** Second, Java is a relatively clean language in terms of semantics. The absence of things like explicit memory management or gotos makes adding Paralocks to it much easier, not to say at all feasible.

**In the footsteps of giants** Third, Java is also the base language that underlies Jif, the only (other) full-fledged security-typed programming language to date. Jif is at the same time a competitor as well as an obvious source of inspiration for Paragon. By basing Paragon on Java too, not only do we make it easier to compare and analyse differences between the two languages, but we can also directly use much of the existing ideas and research that has been done in the context of Jif, e.g. for runtime policies ("labels" in Jif), when implementing Paragon. On top of this, the dominant position held by Jif within the community for language-based security research for the past decade means that our fellow researchers within this domain are already used to working in, and relating to, Java. We hope that this will lead to an easier adoption of Paragon as a platform for future research in this field.

For the wider picture, however, it is far from certain that Java is the optimal host language for a Paralocks-based security-typed programming language. There are several reasons why Java – and the standard Java implementation in particular – is not perfectly suited for the task. These reasons range from small, amendable issues like non-opaque pointers and static variable initialisers, to more conceptually difficult issues like information flow channels introduced by the thread scheduler.

In order to fully fix all these issues, it is likely that a new language would need to be built from scratch, considering such issues at the core design stage.

### 4.1.2 Design Guidelines

The aim here is to make Paragon a strong, expressive, practical and useful language, that is at the same time tractable to program in. Adding security typing invariably requires introducing annotations for the policy type checker, thus adding more syntactic "noise" to an already verbose language. One clear design aim is therefore to keep these annotations as little intrusive as possible, to keep the language tractable. Preferably this means to make annotations as few as possible, as intuitive as possible, and the least aesthetically disruptive as possible. Naturally this aim affects all levels of design, from syntax through semantics of the new language constructs, to issues of policy checking and inference.

Another design principle is to make the language as complete as possible with respect to information flow aspects, but not necessarily as complete when it comes to supporting all the various aspects of Java. Thus, while we would naturally prefer to support as much of Java as possible, we will at this stage unashamedly accept presenting a (sizable) subset of Java that still allows us to support all aspects of Paralocks.

## 4.2 Example Programs

In this section we introduce Paragon through a series of encodings of various information flow policy mechanisms. We present them by implementing each mechanism as a library, which serves a two-fold purpose. First, these examples allow us to introduce the features of Paragon step by step. and put both the basics and the more intricate parts into context. Second, it lets us demonstrate the generality of Paragon as an implementation language for a large variety of different policy mechanisms, and how, by the use of encapsulation, we can present each mechanism through a consistent interface.

### 4.2.1 Simple Declassification

Our first example is just a simple declassification mechanism showing succinctly how class encapsulation gives us the possibility to encode a policy scheme as a library.

The interface of this scheme consists of three things: policies for data that is secret ("high") and public ("low") respectively (not to be confused with Java's notion of "public", i.e. exported from a class), and a method `declassify` that takes secret data as input and releases it as public.

First we define the policy `low` as the least restrictive policy, for data that anyone can see:

```
public static final policy low = { 'x: };
```

Policies in Paragon are first class values of a primitive type **policy**. This way we allow for policies that are not known until runtime (c.f. *runtime labels* [ZM07a]), further discussed in section 4.3.9. For a policy to be used to annotate a variable, we require that policy to be marked **final**, i.e. immutable. This ensures that the policies remain consistent throughout the program.

For instance, two variables annotated by the same policy name are guaranteed to really have the same policy, as there can be no redefinition of that policy name between their declarations.

We change the syntax of policies slightly from Paralocks, to emphasise the policy head and to give more of a Java feel. A policy is still a set of clauses separated by semi-colons, but each clause is written with the head first, possibly followed by a colon and a list of conditions (locks) to be fulfilled (open) for the actor in the head to observe data annotated with this policy. Polymorphic actors are marked with a preceding $'$, which means we no longer need to explicitly quantify actors. The policy above, `low`, is thus equivalent to the Paralocks policy $\{\forall x.x\}$, i.e. bottom.

High data may be made visible to low observers through declassification. We represent this with a condition (lock) `Declassify`:

```
private lock Declassify;
```

Unlike policies, locks are not first class values in Paragon, and cannot for instance be stored in variables. Locks are always implicitly **static**, to avoid aliasing problems. We discuss aliasing issues in more detail in section 4.3.4.

The policy `high` is now simply that which specifies that the data may be made visible to a low observer when the lock is open:

```
public static final policy
    high = { 'x : Declassify };
```

This is equivalent to the Paralocks policy $\{\forall x.Declassify \Rightarrow x\}$.

The act of declassification then becomes a simple matter of taking data with policy `high` and, in a context where `Declassify` is open, *reannotating* it with policy `low`. Such reannotations typically happen at assignments, but can also happen at e.g. the return of a method. This is exactly what the method `declassify` does:

```
public static ?low <A>
  A declassify(?high A x){
    open Declassify { return x; } }
```

There are several interesting things to note about this method declaration. First, it shows how to use policy annotations in Paragon: We simply extend the list of possible *modifiers* on e.g. variables and methods. Here we see that the formal parameter `x` has a modifier `?high`, stating that an argument to the method should have a policy no more restrictive than this. The method itself has a modifier `?low`, denoting the *return policy*, i.e. the effective policy on data returned by the method.

Another thing to note is that neither of the policies we declared, nor the lock, were annotated with policies. They still have policies though: for fields the default policy is bottom if nothing else is specified. For locks, it is top.

Also, the body of the method consists of a single statement: a *scoped* **open** statement. The scoped **open** keeps the specified lock open for the extent of its body. In other words, it opens the specified lock at the start (if it was not already open), closes it when done (unless it was already open at the start), and rules out any (non-scoped) opens or closes of that lock throughout the body.

Finally, as suggested above, returning from a method causes a reannotation of the returned data to the declared return policy of the method. Here the reannotation is valid since it appears in a context where `Declassify` is open.

We also note that this method is now the *only* way to declassify data from `high` to `low`, since the `Declassify` lock is declared to be private to this class. Our library can thus have a simple, consistent interface through the use of standard encapsulation techniques.

To further show the strength, we could easily extend our library with the notion of data that may *never* be declassified:

```
public static class DeclassWithTopSecret
extends Declassification {
  public static final policy top = {};}
```

Data annotated with policy `top` can never be the argument to `declassify`, since that method's parameter is stated to be no more restrictive than `high`.

### 4.2.2 Robust Declassification

In section 3.6 we discussed how to encode *robust declassification* [MSZ04] using Paralocks. We can now implement our encoding as a library in Paragon.

To achieve *robust declassification* [MSZ04], we need to introduce data integrity policies. Integrity is the dual of confidentiality, and we can handle the two concepts in just the same way. Robust declassification requires that the choice of what data to declassify cannot be affected by *untrusted* data.

To model integrity, we need an actor representing a user who *only* observes, and acts on, trusted data:

```
private static final actor trustor;
```

Actors, like policies, are values of a primitive data type, **actor**. This data type is special in that it allows no literal values, and there are no expressions that create new actors. Instead, all variables of type **actor** that are declared without an explicit initialisation get assigned a value implicitly, just like for other primitive types in Java. The difference here is that there is no one default value (like e.g. 0 for **int**), instead each value assigned this way will be unique to each declared variable, thereby creating fresh actor IDs.

By making the actual values inaccessible to the program – i.e. actor identities are opaque – we will not need to worry about any information flow leaks that this generation of fresh identities could otherwise cause.

Values of type **actor** are first class values. Just like for policies, for an actor to be mentioned in annotations on data that actor must be declared **final**, to ensure consistency.

With actor `trustor`, we declare the policy of trusted data as:

```
public static final policy
  trusted = { trustor: };
```

Now we wish to combine the notion of trusted data with the simple declassification mechanism above, but then we run into a problem. Since we modeled `low` as bottom, which is the least restrictive policy possible, all data marked with `low` would be implicitly trusted – i.e. observable by `trustor` – already! Our formulations of the policies in our previous library were too simple to be combinable with the notion of integrity as required for robust declassification. We need to define new versions of `low` and `high` based on an explicit observer separate from `trustor`:

```
private static final actor observer;
public static final policy
  low  = { observer: },
  high = { observer: Declassify };
```

Now we can modularly form the combinations we need, e.g. trusted *and* low data would have the policy `trusted` ⊓ `low`.

To ensure robustness, i.e. that the choice whether to declassify is not based on untrusted data, we give the lock that governs declassification the policy `trusted`.

```
private static ?trusted lock Declassify;
```

The policy checker will then ensure that this lock cannot be manipulated in contexts that depend on untrusted data. The type of `declassify`, which manipulates the lock, will reflect this in its *write effect* modifier, written with a leading !:

```
!trusted ?(low ⊓ policyof(x))
public static <A>
  A declassify(?high A x) {
    open Declassify { return x; } }
```

Indeed the same holds for other data marked as trusted – it cannot be affected by untrusted data, either explicitly or implicitly.

Notice how the method is polymorphic in whether its argument is trusted or not. The parameter is marked with policy `high`, which really means "no more restrictive than `high`", so arguments to the methods could be trusted, untrusted, or even low if we wanted.

The return policy, `?(low ⊓ policyof(x))`, states that the result will have the same policy as the input, only it will now (definitely) be low (`low ⊓ high` is `low`).

The **policyof** operator is a built-in Paragon primitive that can be used only in the specification of policy modifiers for methods. It takes as its argument a parameter of the method, and returns the policy of the argument supplied for that parameter. In the above example, **policyof**`(x)` will thus not return the policy of parameter `x` (which is specified to be `high`), but rather the policy of whatever argument is passed to the method (which we know will be no more restrictive than `high`).

This example also shows that the parameters of a method are in scope when declaring the modifiers for that method.

This library could easily be extended with a mechanism for *endorsement*, similar to declassification.

### 4.2.3 Sealed-bid Auctions

Our next example is not a library but an actual application, the same example we used as a motivation in section 2.2: a server for running online sealed-bid auctions. In this setting we want to model the following information flow properties:

- bidders provide sealed bids and can see their own bid, but cannot see each others' bids.

- bidders learn of the winning bid, but only at the end of the auction.

We only sketch the implementation of the system here, focusing on the parts that are interesting from a Paragon perspective, leaving many other things underspecified.

A bidder is represented in the system as an actor. The bid placed by actor `a` should be visible only to `a` while the auction is running, and be released to all other bidders when the auction is complete assuming it was the winning bid. We model this with the policy `{a; 'x: AuctionClosed, HasBid(x) , Winner(a)}`, where we assume the existence of the lock `AuctionClosed` and the two unary lock families `HasBid` and `Winner`, with intuitive interpretations.

We wrap a bidder and their associated information and operations as a class `Bidder`, starting with the following:

```
final actor id;
final policy bidpol = {id: ;
  'x:AuctionClosed,HasBid('x),Winner(id)};
?bidpol int bid;
```

We implicitly also assume a channel, `chan`, over which to communicate with the actual bidder.

We note that the actors here, unlike those used in the previous examples, are not marked as **static**. This means that each instance of `Bidder` will have a separate actor, uniquely generated when that instance is created. Since actors also affect *typing*, as *singleton types*, aliasing of objects that contain actors becomes a problem. We discuss this in more detail in section 4.3.4.

When the bidder supplies a bid as requested, we signal this by opening the corresponding `HasBid` lock. If the bidder fails to supply a bid, we throw an exception:

```
+HasBid(id) !bidpol
void getBid() throws !bidpol NoBidExc {
  bid = chan.get(); open HasBid(id); }
```

Two things are worth noting here. The first is the `+HasBid(id)` modifier, which signals to the type checker that calling this method will open that lock, assuming the method call terminates normally. If it instead terminates with an exception, we make no such guarantees. The second thing to note is the write effect modifier on the declared exception. Roughly speaking, this policy denotes the level at which it will be possible to observe that the function has terminated with this exception. Java does not normally allow modifiers on declared exceptions – they are an addition in Paragon.

Running the auction now consists of four phases: Getting the bids from all the bidders, determining the winner, reporting the results, and handing out the spoils. The first phase simply loops over all bidders, gets the bid of each, catching exceptions along the way:

```
!bottom void collectBids() {
  for (Bidder b in bidders) {
    try { b.getBid(); }
      catch (NoBidExc e) {} }}
```

The only thing to note here is that the contents of the set `bidders` must be observable by all the bidders, due to the write effect of `getBid`. The same is true for the overall write effect of this method – every bidder can observe that the method has been called, so the only sensible write effect policy is bottom.

In the next phase we look at all the collected bids, determine the winner among them, and declare the auction closed. We first declare a policy `allBidders` as the part of the policy on bids that is not specific to a particular bidder:

```
final policy allBidders =
  {'x: AuctionClosed, HasBid(x)};

+AuctionClosed ?allBidders
  Bidder determineWinner() {
    Bidder winner;
    for (Bidder b in bidders) {
      if (HasBid(b.id)) {
        if (winner == null
            || b.bid > winner.bid) {
          winner = b; }}}
    open AuctionClosed;
    return winner; }
```

The local variable `winner` must have policy `allBidders` for the above code to be type correct. We don't need to explicitly annotate it with that policy though – Paragon performs inference of policies for local variables.

Also noteworthy is that the assignment to `winner` does not affect the write effect of this method, since `winner` is only available locally within the body of the method, so changes to it will not be visible from outside a call to the method.

The method is guaranteed to open the `AuctionClosed` lock, as signaled by the appropriate modifier.

Next we want to notify the bidders about the winning bid:

```
~AuctionClosed !bottom void
  reportResult(?allBidders int winBid){
    for (Bidder b in bidders) {
      if (HasBid(b.id)) {
        b.chan.put(winBid); }}}
```

We assume that the channel to `b` makes the data sent on it available to `b`, i.e. it can only output data with policy (no more restrictive than) `{b.id:}` . To be allowed to send `winBid`, with policy `allBidders`, on this channel, we must know that we are in a context where the two locks mentioned in that policy are truly open. The modifier `~AuctionClosed` declares that this method *expects* that lock to be open whenever it is called. Calling it in a context where that lock is not guaranteed to be open is a type error, and consequently the body of the method may assume that the lock is indeed open. For the second lock, we rely on so called *runtime querying* for the status, through an **if** statement. If the condition of the **if** is a lock, the type checker can assume that that lock is open when checking the *then*-branch. Thus the reannotation of `winBid` is correctly allowed.

Tying all these pieces together we could now write the main code as follows:

```
getBids();
Bidder winner = determineWinner();
if (winner != null) {
  open Winner(winner.id);
  reportResult(winner.bid);
  sendSpoils(winner); }
```

where we leave to imagination how `sendSpoils` should be annotated and implemented, but surmise that it requires the appropriate `Winner` lock to be open. We note that the reannotation of `winner.bid` is allowed when using it as the argument to `reportResult`, since we know that `Winner(winner.id)` is guaranteed to be open.

In this section we introduced several new concepts, each of which is presented in more detail in a later section: Lockstate modifiers (section 4.3.1), runtime querying of locks (section 4.3.2), exceptions (section 4.3.6), policy inference (section 4.3.8), and instance actors and aliasing (section 4.3.4).

### 4.2.4 Lexically Scoped Flows

Our next example is an encoding of lexically scoped flows, reminiscent of the work by Boudol and Almeida Matos [AB05]. The basic idea is a language mechanism introducing allowed flows between security levels (actors, principals) inside a lexically enclosed scope:

```
flow (x to y) { ... }
```

In this example, within the scope of the enclosed block data owned by (or that could flow to) principal $x$ may also flow to principal $y$.

We will first show how to *encode* the mechanism, which in itself requires some new features of Paragon. Then we go on to show how to *encapsulate* our encoding, to present the interface we want.

To encode this scheme we need a lock family with two parameters, where each lock in the family represents a flow relation between its two actor arguments. In this scheme, the relation of flows between actors is transitive and reflexive. We can make our lock family model this using lock *properties*, which allow us to specify conditions under which some locks in the family are *implicitly* open:

```
public lock Flow(from,to) {
  Flow('x,'x);
  Flow('x,'y): Flow('x,'z), Flow('z,'y);}
```

The lock properties must have the correct lock family in the head, but could mention other locks among the premises (we show one such example in section 4.6.4).

The parameters `from` and `to` in the declaration are solely used to specify the arity of the lock family. The names themselves are purely mnemonic.

Since transitivity is a common property, Paragon has a short-hand modifier for it, along with modifiers for reflexivity and symmetry. Our lock definition could thus be written more succinctly as:

```
public reflexive transitive
  lock Flow(from,to);
```

Note that these property short-hands work only for lock families representing binary relations, i.e. with arity 2.

With this lock family we can easily encode the flow mechanism using scoped **open** statements, so e.g. `flow (x to y){ ... }` would be encoded as **open** `Flow(x,y){ ... }`.

Finally we need to encode the policy annotations used with this scheme. If some data is owned by some actor, that data should be allowed to flow to other actors as well when the proper flows are enabled. This means that data for actor `a` should be encoded as `{ 'x: Flow(a,'x)}` (which through reflexivity includes `a`).

The three components we have defined here – the `Flow` lock family, the encoding of the flow statement, and the encodings of policies – is all we need to be able to express the idiom.

**Encapsulation** The encoding above suffers from a clear drawback: it does not present a consistent interface. Specifically it requires the `Flow` lock family to be exported for use in the scoped **open** statements as well as in the encoded version of policies, but that also opens up for unintended uses such as non-scoped versions of **open** or **close**. It also requires a programmer to know how to write the encoded policies, which, while not an immense burden for so simple policies, requires knowledge about the internals of this policy encoding. These problems are classic symptoms of a lack of encapsulation.

A properly encapsulated version of the library would make the lock family internal, and export only two things: a way to construct consistent policies, and a method representing the flow construct.

To enable encapsulation of policy construction, Paragon allows so called *type methods*. These are static, pure methods that can be deterministically evaluated by the policy type checker at compile time. We mark them with a special modifier **typemethod**. The type method needed in our example is one that takes an actor and produces a policy for that actor's data:

```
public typemethod policy pol(actor a) {
  return { 'x: Flow(a,'x) } }
```

Some data owned by actor `a` could now be annotated with policy `?pol(a)`, which the Paragon type checker can replace by the result of evaluating the method.

To properly encapsulate the flow construct, we run into a problem with our choice of host language: Java does not have first class statements or procedures (let alone functions). To be able to pass to our envisioned `flow` method the code to run as its body, we need to use a standard Java trick and wrap that code up as a method of an anonymous object implementing a declared interface, and then pass that object as the argument.[1]

Our `flow` method then needs to take three arguments: The two actors for which to enable the flow, and the object representing the body. However,

[1]Had our host language been e.g. Scala instead, this would not have been an issue.

the two actors are not normal arguments, since their intended purpose is to affect the *typing* of the body, which will be the third argument. Thus we need to pass the actors as *type arguments* to the method:

```
public static <actor A, actor B>
  void flow(FlowBlock<A,B> block) {
    open Flow(A,B) { block.go(); }}

public interface
  FlowBlock<actor A, actor B> {
    public ~Flow(A,B) void go(); }
```

As seen by the `~Flow(A,B)` modifier, the inner code may depend on this lock being open, enabling flows from `A` to `B` in the body as intended.

Note that the type parameters specify that they in turn have type – or *kind* – **actor**, to distinguish them from ordinary Java type arguments, which conceptually and implicitly have kind *type*.

We could now put these components to use together as follows:

```
flows<a,b>(
  new FlowBlock<a,b>{
    public ~Flow(a,b) void go(){ ... }});
```

This encoding is unfortunately far more verbose than the mechanism we are encoding – but we note that all the extra noise comes *solely* from Java not having first class statements.

The perceptive reader may have noticed a problem with our encoding, or perhaps even two, albeit slightly related. The first issue is to do with side effects. We did not specify a write effect policy on either `flow` or `go`, which means they will both default to `{}`. This in effect means that the body of an implementation of `go` cannot have side effects, or at least none visible at policies lower than `{}`.

Preferably we would like `flow` to be polymorphic in the write effect of its argument. Again we turn to type parameters, this time passing the intended write effect to the interface as a type argument of kind **policy**:

```
public interface
  FlowBlock<actor A, actor B, policy W> {
    public ~Flow(A,B) !w void go(); }

public static !w
  <actor A, actor B, policy W> void
    flow(FlowBlock<A,B,W> obj) { ... }
```

Now the body can perform arbitrary side effects as far as `flow` is concerned. However, it now appears that a programmer using this interface would need to explicitly specify the write effect as well, which would be unfortunate. To avoid this in most cases, Paragon attempts to infer the type arguments when left unspecified, which in the current case would be straightforward to do for the write effect policy argument. This is similar to how Java infers type arguments of generic methods.

Our usage example could thus be written

```
flow<a,b>(new FlowBlock<>() {
  void go() { ... }}
```

where we let the type checker infer all the type arguments for the `FlowBlock` constructor, as well as the policy argument to `flow`.

The second issue relates to the lock state. As seen by the signature of the method `go`, the body of an implementation of that method could *only* depend on one specific lock being open, even if it was called inside the scope of several nested `flow` calls. We need a way to make the methods *context sensitive* with respect to the lockstate. The pattern is familiar by now: we let the methods take yet another type parameter, this time of kind **lock**`[]`:

```
public interface FlowBlock
 <actor A, actor B, policy W, lock[] L>
   { public ~Flow(A,B) ~L !W void go(); }

public static !W ~L
 <actor A, actor B, policy W, lock[] L>
   void flow(FlowBlock<A,B,W,L> obj)
     { ... }
```

Again we rely on inference to fill in the proper lockstate context, leaving no more to write for a user of this library than before.

With these definitions of `pol`, `flow` and its auxiliary interface `FlowBlock`, we have achieved a proper encapsulation, ensuring that the library is used consistently. While the overhead for the programmer is somewhat larger than we would prefer, that overhead is really an artifact of Java, not Paragon.

Lock families and properties are discussed in more detail in section 4.3.2. Type parameters are discussed in section 4.3.3, type methods in section 4.3.5, and type inference in section 4.3.8.

## 4.3 The Paragon Language

In this section we take a more formal approach to the features of Paragon. The examples of the previous section have given a flavor of the language and its features, aimed for the casual reader or as a first introduction. In this section and the next we offer a more detailed account of the design and implementation of the language.

The remainder of this section is structured as a reference to the various features, each presented in a separate subsection, with no overall narrative. We will cover the following sections in turn: Types, policies and modifiers (4.3.1); Locks (4.3.2); Type parameters (4.3.3); Actors and aliasing (4.3.4); Type methods (4.3.5); Exceptions and indirect control flow (4.3.6); Field initialisers (4.3.7); Type and policy inference (4.3.8); Runtime policies (4.3.9).

### 4.3.1 Types, Policies and Modifiers

In Paragon every information container (field, variable, lock) has a policy detailing how the information contained therein may be used. Every expression has an effective policy which is (an upper bound on) the conjunction of all policies on all containers whose contents are read by it – we refer to this as the expression's *read effect*. Similarly every expression (and statement) has a *write effect*, which is (a lower bound on) the disjunction of all policies on all containers whose contents are modified by the expression.

Paragon (unlike Jif – see section 4.6.3) separates policies from base types syntactically by having all policy annotations as modifiers. All in all, Paragon adds nine new modifiers over Java. Two of them relate to policies:

- `?pol` denotes a policy on an information container, and the read effect of accessing that container. When used on a method we refer to it as the *return policy*, as it is the read effect on the value returned by the method.

- `!pol` denotes a write effect, and is used to annotate methods. They are also used to signal the write effects of thrown exceptions (see section 4.3.6) and of static initialisers (section 4.3.7).

There are also three modifiers used only on methods to detail their interaction with the lockstate:

- `+locks` says that the method *will* open the specified lock(s), for every execution in which the method returns normally. We call this the *opens* modifier.

- `-locks`, dubbed the *closes* modifier, says that the method *may* close the specified lock(s), for *some* execution.
- `~locks`, the *expects* modifier, says that specified lock(s) must be open whenever the method is called.

The *opens* and *closes* modifiers are also used for exceptions, discussed in section 4.3.6.

The other four modifiers introduced by Paragon are the three short-hand modifiers for lock properties discussed in section 4.3.2, and the **typemethod** modifier discussed in section 4.3.5.

### 4.3.2 Locks

Locks in Paragon are not first class. They cannot be stored in variables, nor can they be passed as arguments to methods. The only way to manipulate the status of a lock is via **open** and **close** statements. However, the status of a lock may be queried at runtime, in ways such that it makes sense to treat **lock** as a type.

If a lock is used syntactically as an expression, the type of that expression is considered by Paragon to be **lock**. If an expression of type **lock** appears as the condition of an **if**, **while** or **do** loop, or as the first operand of the ternary conditional operator `?:`, the type checker can assume that the lock is open when checking the branch corresponding to the condition being satisfied. Apart from this effect on the typing of programs, all expressions of type **lock**, in conditions or elsewhere, are implicitly cast to **boolean**.

Paragon allows methods to specify **lock** as their return type. This is particularly useful for implementing "getter" functions, to avoid exporting the locks themselves for manipulation outside a class. The body of a method with return type **lock** must consist only of a single statement **return** *e*, where *e* is an expression of type **lock**.

Finally we also allow the use of the operator `&` on locks. If an expression of type **lock** that includes `&` operators appears in a condition, the type checker can assume that all the operand locks are open in the appropriate branch.

**Lock properties** Lock families can be declared to have *properties*. A property specifies conditions under which some locks in the family are *implicitly* open. A concrete lock can thus be explicitly closed, but still remain open due to some property, such as transitivity, keeping it open implicitly.

Lock properties are a modest restriction of the full potential power of *Recursive Paralocks*, discussed in section 3.7. Specifically the properties must be declared at the point where the lock family is declared, so a family either has a property or not. It is not possible in Paragon to e.g. turn transitivity on and off dynamically during the execution of a program.

Policy checking in the presence of recursive clauses is a far more challenging problem than without them, as discussed in section 3.7. Essentially the policy language with recursive clauses is equivalent to DATALOG and runtime querying for locks is equivalent to evaluating DATALOG queries. The more difficult part is entailment of policies, which is equivalent to the problem of containment of a non-recursive DATALOG program in a recursive one. This problem is EXPTIME-complete in general, though in most cases our policies do not require the full generality. Also our saving grace is that, at worst, we can ask the programmer to insert extra annotations to avoid inference in cases where it would otherwise take too long.

### 4.3.3 Type Parameters

Java, since the introduction of "Generics" in Java 5.0, allows types and methods to be parametrised by types, giving Java parametric polymorphism. Paragon introduces several new entities – actors, policies and locks – that affect typing in various ways. It is natural to extend the polymorphism to also include these aspects. The different entities are clearly not interchangeable, which implies the need for a *kind* system for type-level entities.

Thus in Paragon ordinary types have the implicit kind *type*. Type parameters of kind *type* need not be annotated, like in vanilla Java. For the Paragon-specific entities we introduce *kind annotations*, to separate them from each other and from ordinary types. For actors and policies we can simply reuse their types as kinds as well. We can do the same for locks, though, as shown in the example in section 4.2.4, we need to be able to parameterise over not just single locks, but rather sets of locks. To avoid introducing new keywords, we reuse the syntax for arrays for this purpose, i.e. the kind annotation on parameters taking sets of locks is `lock[]`.

With Generics, the Java type checker tries to infer type arguments where none are provided. Starting in Java 7 this works for both methods and constructors. Paragon does the same for missing type arguments, with a slight generalisation – we allow partial type argument lists, which are assumed to instantiate the type parameters of the method or constructor from left to right. We saw an example in section 4.2.4, where we supplied the two actor arguments but left the policy and lock set arguments to be inferred.

### 4.3.4 Actors and Aliasing

Actors and locks together play a crucial role in the typing of Paragon code. Locks determine what flows are allowed at what points, and locks are often parametrised by actors. The typability of some code may depend on a given lock, with some given actor arguments, being open. Formally, the type checker treats actors as *singleton types* [Asp95].

However, the possibility of aliases greatly complicates things. If some code opens lock `L(a)` and then closes lock `L(b)`, is the first lock still open? Clearly that depends on whether `a` and `b` are two different actors, or aliases of the same actor.

Alias analysis in Java is a well-studied area, with many possible degrees of sophistication. For Paragon, erring on the side of caution is clearly crucial, so any analysis that *conservatively* approximates actor aliasing would be adequate.

For static actor fields, and for actor variables declared in the current method, Paragon tracks what variables are known to be distinct or aliases, and for which variables we simply cannot say. We take the conservative approach, and assume closed all locks that could potentially be affected by a **close**.

For *instance actors*, i.e. non-static fields of type **actor**, the situation is slightly more complex. To track the contents of actor field `obj.a`, we not only need to consider possible aliases of the actor itself, but also possible aliases of `obj`, which could cause `obj.a` to change without ever being syntactically updated.

To avoid a full-scale alias analysis over *all* fields and variables (which would not be feasible), Paragon always treats all instance actors belonging to the same class as potential aliases. In practice this means that we don't disallow programs, but may require them to use more runtime queries to compensate for weaknesses in the alias tracking. Another option could be to instead restrict the creation of aliases through e.g. affine typing of classes with actor member fields. Doing that would rule out quite a few interesting programs though, among them the sealed-bid auction example given in section 4.2.3. We naturally prefer to allow such programs, and rather require the extra runtime query where needed, as seen in that example.

Locks are not first-class and thus cannot be directly aliased. By making all locks static we also rule out the possibility of indirect aliasing that instance actors suffers from. We note that this restriction does not limit expressiveness, since a non-static lock can easily be mimicked by a static lock and an extra non-static actor argument. Thus we reduce the aliasing problem to actors only.

### 4.3.5 Type Methods

A type method is Paragon's name for methods that can be evaluated by the type checker at compile time, in order to determine policies on variables, fields and methods. A more formally correct name would perhaps be type *functions*, since these methods must be both pure, i.e. have no side-effects, and deterministic. By deterministic we mean that the end result may only depend on values known statically when the method is called. That includes the method's arguments, as well as certain static fields. For a field to be useable in a type method, it must be static and final, have a primitive type, have a policy bottom, and have a simple initialiser that can be evaluated at compile time without side-effects.

The fact that type methods can be used at compile time does of course not preclude them for being used at runtime as well, where they behave like static methods.

### 4.3.6 Exceptions and Indirect Control Flow

The static policy type system in Paragon tracks two kinds of information flows: direct flows arising from assignments, and indirect flows arising from control flow. It makes no attempt to track flows arising from termination – it is *termination insensitive*. If exceptions could not be caught, an exception would be the same as (premature) termination, which means we would not need to care about them. However, the catch mechanism makes exceptions rather a kind of control flow primitive, needing special attention.

In Java, subclasses of `RuntimeException` are *unchecked*. This means that methods need not declare if they could terminate with such an exception. Examples of runtime exceptions are `ArithmeticException` which for instance arises from division by 0, `ArrayOutOfBoundsException` and `NullPointerException`.

It should be obvious that any exception that can be caught is a potential channel for information flow, which means that in Paragon all exceptions must be checked. This in turn implies the need for analyses that can rule out the possibility of exceptions, in particular a null pointer analysis is needed to avoid that every instance field or method use incurs the need to declare a possible `NullPointerException`.

A caught exception is in essence a jump, where control is passed from the throw point to a catch block. Such a jump may be noticeable by anyone who can notice either the catch block being executed, or the statements in the normal control flow past the throw point. To avoid unintended flows, all such statements must be constrained by the context in which the throw appears. We refer to this as the exception's *area of influence*.

Since an exception might not be caught locally, the area of influence is not a local property in general. However, the area of influence of a **throw** statement cannot in general be calculated as a local property. A method that includes a **throw** might not catch the exception, but instead declare that it **throws** the exception for a caller to handle. To accurately compute the write effects of **throw** statements in this general case would require whole-program analysis, killing modular compilation. Clearly this is not an option.

Instead we let methods that throw exceptions *declare* the write effect of those exceptions, which then becomes a lower bound on the writes allowed in the exception's area of influence. This write effect is declared as a modifier on the proper exception type in the method's **throws** clause.

This declared write effect serves as an approximation of the context where the **throw** appears. It is thus both used as the effective write effect of the **throw** statement itself, to ensure that it is not used in even more restrictive contexts, as well as a bound on the write effects of all statements in the area of influence.

Constraining subsequent statements in this way is reminiscent of the way termination sensitivity is typically achieved for information flow type systems. Indeed, the constraints we introduce here *will* stop termination leaks caused by uncaught exceptions. To achieve full termination sensitivity however, we would also need to disallow "low" effects to follow potentially non-terminating "high" loops. We choose not to do so though, for two reasons. First, the mechanisms needed to perform the required analysis would be pervasive and heavy-weight (all methods would need the equivalent of a termination effect). Second, since every loop in the program would be like a potentially thrown exception that can never be caught, putting large constraints on subsequent statements, it would significantly reduce the class of type-correct programs.

Since uncaught exceptions are effectively premature exit points from a method, the *opens* and *closes* modifiers pertaining to a normal exit do not apply when entering a catch block. Hence we let the declared exceptions also take opens and closes modifiers, specifying the lock state that will be in effect at the start of a corresponding catch block.

For the cases where a thrown exception is caught locally, before ever reaching the top level of a function, there will be no need for approximations via declared policy or lockstate modifiers. Instead all the necessary information can be computed locally.

Interestingly, several other control flow mechanisms in Java can be treated as special cases of exceptions for purposes of policy inference: **return**, **break** and **continue**. These are simpler to handle than exceptions, since their area of influence is always contained locally.

For **return** statements, the declared return policy of the method serves as both an upper bound on expressions used as arguments to the **return**, and as the effective (local) write effect of all **return** statements. Methods returning nothing (i.e. with a **void** return type) would have no declared return policy. Instead the effective write effect can be approximated as the current PC, since that would be the limit of what would be allowed at the point of the **return**, while putting the least constraints on statements in the area of influence. The area of influence of a **return** statement naturally extends to the end of the method body.

Handling **break** and **continue** statements is very similar to handling **return** statements with no arguments. Both can use the current PC as their effective write effect. For **continue**, the area of influence extends to the remainder of the *body* of the loop it is contained in. For **break**, the whole of the loop, including its condition or preamble, is affected, just like for exceptions thrown in loops. **break** statements can also end **switch** statements, in which case the area of influence covers all subsequent statements.

### 4.3.7 Field Initialisers

Initialisers for fields are simply expressions, and quite naturally the effective policy on such an expression cannot be more restrictive than that on the field. But expressions also have a write effect – i.e. the initialisation taking place might cause visible changes elsewhere. Furthermore, an initialisation could potentially fail with an exception.

Fields come in two different flavors: static fields and instance fields. For both, the main difficulty lies in handling their initialisation – their side-effects and possibility for exceptions – and the solution differs between the two.

**Instance fields** are all initialised when the instance they belong to is created. This means that we can view the initialisation code as being an implicit prelude to every constructor for the class. The solution is then natural – the write effect of the initialisers cannot be less restrictive (more revealing) than the declared write policy on any constructor. Similarly, if an initialiser could throw an exception, all constructors for the class must declare this.

Note that the policies on the fields themselves do not contribute to the write effects of constructors, as there is no way that the contents of the fields could give away the *existence* of the object they belong to, i.e. that the constructor was executed. Anyone having access to read those fields must obviously already know that the object exists.

**Static fields** have the same issues with write effects and exceptions, but the story is far less simple. In Java, all static fields of a class are initialised at the same time, whenever any one of them is used in the program. This in effect means that *any* use of a static variable will have the worst-case write effect of all the static initialisers for the same class. We can analyse whether a static initialisation is guaranteed to already have taken place before a given use of a static variable, to preclude it from carrying the write effect.

Regardless of whether a particular use of a static variable should be assigned a write effect or not, we need to notify the type checker of such write effects. For instance fields, the write effect is naturally declared on the constructors. For static initialisers, which share a single overall write effect, we could imagine two choices for the placement of a write effect modifier, neither very orthodox. The first is that each static field in a class should reports the write effect that is a lower bound on the write effect of all static initialisers. In other words, every static field must have the same write effect modifier, causing redundancy. The second option, which we have chosen for Paragon, is to place that modifier among the modifiers for the class. This means that the write effect modifier only needs to be specified once, and it is more intuitively clear that this write effect is an upper bound on the write effects of *all* the initialisers of static fields of that class.

Regarding exceptions, it is even more difficult to find a place to declare if a static initialiser may throw one. In Java, if any static initialiser fails then the whole class fails to initialise, and, regardless of which type of exception that caused the failure, the result of the field use that caused the attempted initialisation is a an `ExceptionInInitializerError`. Any subsequent attempt to access a field of the same class, assuming the first error is caught (which Java discourages for subclasses of `Error`), will result in a `NoClassDefFoundError`.

We could possibly imagine a similar solution to this problem as to the problem with write effects, letting the *class* declare that it throws these exceptions. This would be a rather ad-hoc change to Java though, compared to adding modifiers for Paragon-centric things, so we choose a more conservative approach. In Paragon, Initialisers for static fields may not fail, as guaranteed by the exception handling methods discussed in section 4.3.6.

### 4.3.8 Policy Inference and Defaults

To reduce the burden on the programmer to put in policy annotations, Paragon attempts to either infer, or supply clever defaults for, policies on variables, fields and functions. When annotations are omitted, the following defaults are assumed:

- Policies on fields default to bottom.
- Write effect policies of methods default to top.
- Return policies of methods default to the join of the policies on all parameters since all of them are expected to contribute to the result.
- If no policy is given for a formal parameter to a method, the method is assumed to be polymorphic in that parameter. The effective policy of a polymorphic parameter `x` is **policyof**`(x)`.
- If no policy is given for a local variable, Paragon attempts to infer the effective policy for that variable.

Policy inference works through a straightforward constraint system, where all constraints arising from comparisons between policies, including the PC, are collected and resolved on a per-method basis. In the general form a constraint will be an inequality between two policy expressions, each of which can contain literal policies, variables denoting policies, and joins and meets.

### 4.3.9 Runtime Policies

Since policies can be used as values at runtime, and dynamically hoisted to the type level, we need ways to relate policies that are not known statically to other (static or dynamic) policies. To achieve this, Paragon needs to perform runtime entailment checks between policies. This problem has been studied by Zheng and Myers in the context of Jif [ZM07a], and we choose to follow their solution.

Similar to runtime lock queries, we thus allow inequality constraints between policies to appear as the condition in **if** statements and conditional `?:` expressions. The type checker can then know when checking the first branch that the inequality holds, and can allow flows that would otherwise have been untypable.

## 4.4 The Paragon Type System

With all the components discussed in the previous section, we can now put them together in a type system. We have not yet formalised the full type system for Paragon, instead we present here a slightly simplified system that covers the core aspects of Paragon typing. In particular, the following restrictions apply:

- We do not cover any aspects of the class hierarchy, e.g. sub-typing, interfaces, overloading of methods, **super**. These aspects are not difficult to handle from an information-flow point of view, so we leave them out for the sake of improving the presentation.

- We do not cover type parameters. Type argument inference for Java is notoriously tricky to formalise, and state-of-the-art formalisations are typically given as descriptions of the inference algorithm, in plain English [GJS96]. We have yet to come up with a suitable formalisation for Paragon.

- Our type system does not include any aspects of exception analysis. We have noted previously that some analyses are crucial to reduce the need to account for null pointers at every use of a variable of a reference type. This is not reflected in the type system presented here.

- We do not cover runtime policies. All policies are assumed to be known statically, and all policy variables are assumed to be final and initialized when they are declared.

- We concentrate on a subset of Java, leaving out a number of features that do not add anything to the presentation. The features left out are enums, static fields, arrays (as in the syntactic sugar), inner classes, casts, most operators, labeled statements, as well as expressions and statements whose typing would be very similar to those already covered (e.g. **do while** is very similar to **while**).

We also note that our Paragon type system only covers information flow aspects of typing, and a modicum of ordinary type checking as needed to properly handle the information flow parts. Our Paragon type system thus expects an ordinary Java type checker to be invoked to handle many ordinary typing aspects.

With this in mind, we can go on to present our restricted Paragon type system.

### 4.4.1 Typing Judgment

The typing judgment for expressions has the deceptively simple form

$$E; S \vdash e : \tau, p \rightsquigarrow S', C$$

where

- $\tau$ is the type and $p$ the effective policy of the expression $e$. We use the convention that upper-case $T$ denotes a class type, while lower-case $t$ denotes any type (including the pseudo-type **void**).
- $E$ is the typing environment.
- $S$ is the type state before evaluating $e$.
- $S'$ is the updated state after evaluating $e$.
- $C$ is the set of policy inference constraints generated by the type rule for $e$.

We will treat $E$ and $S$ as records with named fields. The environment $E$ contains the following fields when checking the body of a class:

- `field` is a mapping from field names to their types and policies. We write $E[\texttt{field}](x)$ for the type and policy of field $x$. We further use pattern matching notation e.g. $E[\texttt{vars}](x) = (t_x, p_x)$ to bind the type and policy of $x$ to the names $t_x$ and $p_x$ respectively.
- `methods` is a mapping from method heads to their signatures. A method head is a name and a sequence of parameter types. We write $E[\texttt{methods}](m(t_1, \ldots, t_n))$ to retrieve a method signature. The signature includes the parameter types since method names may be overloaded[2].

  A method signature is a 7-tuple, consisting of the following: the return type; the return policy; an array of parameter policies; a write effect policy; a set of expected locks; a set of lockstate changes; and a set of exception signatures. An exception signature in turn is a record with three fields: `read` is the read policy on the value thrown, `write` is the write effect of the throw, and `lockMods` is a set of lockstate changes.

[2] Note that since we leave out all aspects of sub-typing, we do not need to bother about the difficulties involved in finding the "best match" among applicable functions.

- `constrs` is a mapping from constructor heads to their signatures. A constructor head is a class type name and a sequence of parameter types, with the same mechanism for retrieving signatures as for methods. A constructor signature is a 5-tuple with the same fields as a method signature, only it has neither return type nor return policy.

- `locks` is a mapping from lock family names to arities and policies.

- `policies` is a mapping from names of policies to the actual policies they represent.

- `types` is a mapping from class names to (restricted) environments. Each class name maps to a record with the five fields listed above, except we use the name `fields` instead of `vars` when reasoning about fields of classes.

- `typemethods` is a mapping from the names of type methods to their compile time behaviour. Specifically, $E[typemethods](m)$ is a compile-time representation of the body of $m$, which can be interpreted during type checking.

- `this` holds the type of the current class, i.e. the type of the expression **this**.

- `types` is a mapping from class names to type environments containing the same seven fields discussed above (and would also contain this field if we considered inner classes).

- `lockProps` is a set (not a mapping) of the lock properties (section 4.3.2) for all locks in scope, thus also including locks in other classes than the one currently being checked.

When checking the body of a method, $E$ will be augmented with the following fields:

- `vars` is a mapping from variable names to variable signatures, i.e. a type and a policy, just like for `fields`.

- `lockstate` is the set of locks known to be open when the method is called, i.e. it is the *base* set of locks that any lockstate changes will be applied to.

- `return` is a tuple holding the return type and return policy of the method.

- `exns` is a mapping from exception types to their read and write policies. We store these as records as discussed for exceptions in method signatures above, only we omit the field for lockstate modifiers. The field will hold mappings for all exceptions with a declared handler – either those that are declared thrown by the enclosing method, or those with enclosing catch blocks.

- `branchPC` holds information regarding the branch PC, i.e. the lower bound on write effects for statements and expressions that is incurred by the context they appear in.

  To account for *local write effects*, the branch PC must be parameterised on which entity that causes the write effect. For example, if a variable is both declared and assigned to inside one branch of an **if**-statement, the effect on the branch PC by the policy of the **if**-statement's conditional expression should be ignored. We thus treat the `branchPC` component of $E$ as a map, from entities to branch contexts, so that e.g. $E[\texttt{branchPC}](x)$ is the effective branch PC calculated only from the branch points where $x$ is already in scope. $E[\texttt{branchPC}](\star)$ gives the full PC. Note that the possible entities in the domain of this mapping are not only the variables, but also locks and exceptions, including the "pseudo"-exceptions **break**, **continue** and **return**.

The state $S$ handles the parts of the analysis that require following execution paths linearly, i.e. the parts that do not follow the block structure. The record $S$ will be used when checking the body of a method, and contains the following named fields:

- `actors` is a mapping from actor variable names to actor information. We will discuss actor analysis in more detail shortly.

- `lockMods` holds (a safe approximation of) the lockstate changes done since the beginning of the method up to the current point in the execution. The actual current lockstate can thus be computed at any point by applying these changes to the base lockstate held in $E[\texttt{lockstate}]$.

- `exns` is a mapping from exceptions to exception points. An exception point represents the point where the exception was thrown, and is a record with two fields: `state` is (a safe approximation of) the state ($S$) at the time when the exception was thrown, and `write` is the write effect policy of the exception.

  At any point during typing, $S[\texttt{exns}]$ will contain mappings for exactly those exceptions for which the area of influence extends over the current

execution point. The current exception PC is the joint influence of all such exceptions, i.e. join of the write effect policies of all exceptions in the domain of $S[\texttt{exns}]$. Formally we define

$$\mathit{exnPC}(S) = \bigsqcup\{S[\texttt{exns}](X)[\texttt{write}] \mid X \in \mathrm{dom}(S[\texttt{exns}])\}$$

**Updating states and environments** We write e.g. $S[\texttt{exns}\{X \mapsto (\texttt{state} = S, \texttt{write} = p)\}]$ for the state where the `exns` field maps exception type $X$ to the record $(\texttt{state} = S, \texttt{write} = p)$, but which otherwise acts as $S$. When the field names are clear from the context, we will simplify the above record to $(S, p)$.

We will further write e.g. $E[\texttt{branchPC}\{a \sqcup= p\}]$ as a short-hand for $E[\texttt{branchPC}\{a = E[\texttt{branchPC}](a) \sqcup p\}]$, and similarly for other operators. We extend this notation point-wise to maps, so that $E[\texttt{branchPC} \sqcup= p]$ means $E[\texttt{branchPC}\{e_1 \sqcup= p, \ldots, e_n \sqcup= p\}]$ for all $e_i \in \mathrm{dom}(E[\texttt{branchPC}])$.

Many fields in $E$ and $S$ must handle block-wise scoping, to allow for variable name shadowing. For the sake of simplifying the presentation, however, we will assume with no loss of generality that all newly declared variables always use names that are not already in scope.

**Actor analysis** Our actor alias analysis records the status of each actor variable in scope. For each variable $a$, $S[\texttt{actors}](a)$ maps to a record with two fields: `id` maps to the identity of the actor, while `stability` holds the *stability* of the variable, discussed below.

If actor variable or field $a$ was declared with no explicit initialiser (and has not been changed since), we know it will hold a fresh actor identity value. Then $S[\texttt{actors}](a)[\texttt{id}]$ will map to $\texttt{fresh}\,k$ for some freshly generated number $k$.

If the value of $a$ cannot be statically known, $S[\texttt{actors}](a)[\texttt{id}]$ will instead map to $\texttt{alias}\,k$, again for a generated $k$. If actor variable $a$ is assigned the value of $b$, it will inherit the identity of of $b$, regardless of whether $b$ is fresh or aliased. We use $\alpha$ to range over actor identities.

Two actor variables are *known* to hold the same value whenever their $\alpha$ values match, regardless of whether they are both fresh or both aliases (one of each can never happen). On the other hand, two actor variables *may* hold the same value if either their $\alpha$ values match, or one (or both) is an alias. This is exactly the same may-alias relation that we used for the type system in section 3.5, and we define it as follows:

$$\begin{aligned} a \simeq b \stackrel{\text{def}}{=} {} & S[\texttt{actors}](a)[\texttt{id}] = S[\texttt{actors}](b)[\texttt{id}] \\ & \vee \mathit{alias}(S[\texttt{actors}](a)) \vee \mathit{alias}(S[\texttt{actors}](b)) \end{aligned}$$

where the function $alias(\alpha)$ returns true iff $\alpha = \texttt{alias}\ k$ for some $k$. We extend this relation to equal-length vectors of actors in a point-wise manner.

Actor fields can be updated indirectly, e.g. the invocation of a method might update some global actor field with no way to signal this to the caller. Thus all non-final actor fields of all objects and classes in scope must be assumed to have changed whenever a method is called, which means our actor map must now map all such fields to fresh aliases. We call such fields *volatile*. Variables local to the current method are not volatile, nor are any fields marked **final**. We all such fields *stable*.

Due to the problem of indirect aliasing discussed in section 4.3.4, non-final instance actors must also be considered volatile in the presence of an update of that actor field for *any* instance of the same type.

We thus have three cases for the `stability` field for variable $a$: `stable`, `volatile` or the field designator $T.f$, where T is the type of the instance to which the field belongs. Whenever we call a method, all fields not marked `stable` will have their identity replaced by a freshly generated $\alpha$ value, unique to each field, and the alias status set to `true`. Also, whenever an actor field $f$ is updated for an instance of class $T$, any field marked with $T.f$ will be scrambled in the same way.

We will write $S[\texttt{actors scrambled}\ T.f]$ to perform the scrambling for field $T.f$, and omit $T.f$ when we want to scramble all volatile fields.

**Combining lockstate modifiers and states** We represent lockstate modifiers as a pair of sets of locks $(C, O)$, where $C$ and $O$ are the locks that have been closed and opened respectively. A lock in either set will be on the form $L(\alpha_1, \ldots, \alpha_n)$ where each $\alpha_i$ is an actor identity. When applying some lockstate modifiers to a lockstate, the intuition is that we first close all the locks in $C$, and then open all the locks in $O$.

We define the operator $\triangleright$ to combine lockstate modifiers in sequence:

**Definition 26** ($\triangleright$)**.**

$$(C_1, O_1) \triangleright (C_2, O_2) \stackrel{\text{def}}{=} (C_1 \backslash O_2, \{L(\vec{\alpha_1}) \mid L(\vec{\alpha_1}) \in O_1, \nexists L(\vec{\alpha_2}) \in C_2 . \vec{\alpha_1} \simeq \vec{\alpha_2}\})$$

There is a symmetry between the two components here that may not be obvious at first glance. The set difference operation could be expanded in the same way as the second part, only using normal equality for comparison instead of our may-alias relation. The intuition is exactly the same as in section 3.5.

We overload the $\triangleright$ operator to also apply lockstate modifiers to lockstates proper, by simply assuming that a lockstate is represented a lockstate modifier pair where the first component is the empty set. We will freely mix notations as suitable for the presentation.

The operator $\triangleright$ handles *sequential* composition of lockstate modifiers. However, in many cases we have parallel potential execution paths, e.g. the two branches of an **if**-statement, and we need to approximate what the lockstate modifiers – or in fact the whole state – will be at the point where the paths merge.

We define the operator $\diamond$ to merge two lockstate modifiers, actor trackers, or exception trackers (i.e. the component fields of the state $S$), and further overload it to merging states by merging the components point-wise.

**Definition 27** ($\diamond$)**.**
For lockstate modifiers: $(C_1, O_1) \diamond (C_2, O_2) \stackrel{\text{def}}{=} (C_1 \cup C_2, O_1 \cap O_2)$
For actor mappings: $A_1 \diamond A_2$ is the actor mapping $A$ such that

$$A(a) = \begin{cases} \alpha & \text{, if } A_1(a) = A_2(a) = \alpha \\ \texttt{alias}\ k,\ k \text{ fresh} & \text{, if } A_1(a) = \alpha_1 \wedge A_2(a) = \alpha_2 \wedge \alpha_1 \neq \alpha_2 \end{cases}$$

For exception mappings: $Ex_1 \diamond Ex_2$ is the exception mapping $Ex$ such that $\text{dom}(Ex) = \text{dom}(Ex_1) \cup \text{dom}(Ex_2)$ and $Ex(X) = (S_1 \diamond S_2, p_1 \sqcup p_2)$, where

$$(S_i, p_i) = \begin{cases} Ex_i(X) & \text{, if } X \in \text{dom}(Ex_i) \\ (S_{ID}, \bot) & \text{, otherwise} \end{cases}$$

where $S_{ID}$ is a distinguished identity of $\diamond$.

### 4.4.2 Typing Expressions

Figure 4.1 contains the typing rules for simple Paragon expressions. The rule for literals appeals to a simple type system for literal values that we exclude here. The type of **this** is simply the one recorded in the environment. The expression **null** can represent a reference of any class type, so $T$ is unconstrained in the rule. Binary operators all follow a simple pattern where the result depends on both operands but the operator itself is pure[3]. This includes the Paragon-specific operators on policies, $\sqcap$ and $\sqcup$[4].

[3] Some care must be given to the `+` operator when given an operand of type `String`. The method `toString()` will then be called on the other operand, and we must ensure that the implementation of this method has no side-effects for the rule given here to hold.

[4] In our prototype implementation of Paragon we use $+$ for $\sqcap$ and $*$ for $\sqcup$.

$$\frac{\vdash lit : t}{E; S \vdash lit : t, \bot \rightsquigarrow S, \emptyset}\ (\text{Lit}) \qquad \frac{E[\texttt{this}] = T}{E; S \vdash \textbf{this} : T, \bot \rightsquigarrow S, \emptyset}\ (\text{This})$$

$$\frac{}{E; S \vdash \textbf{null} : T, \bot \rightsquigarrow S, C}\ (\text{Null})$$

$$\frac{\begin{array}{c} E; S \vdash e_1 : t_1, p_1 \rightsquigarrow S_1, C_1 \\ E; S_1 \vdash e_2 : t_2, p_2 \rightsquigarrow S_2, C_2 \\ t_1 \oplus t_2 : t_{op} \end{array}}{E; S \vdash e_1 \oplus e_2 : t_{op}, p_1 \sqcup p_2 \rightsquigarrow S_2, C_1 \cup C_2}\ (\text{BinOp})$$

**Figure 4.1:** Paragon Type System - Simple Expressions

The rule also holds for the `&` operator when applied to operands of type **lock**, though some complexity is hidden by the seemingly innocuous appeal to the ternary relation $t_1 \oplus t_2 : t_{op}$ in the premise. The type **lock** also tracks *which* lock(s) it represents, so specialising the ternary relation to locks we have that $\textbf{lock}^{L_1} \oplus \textbf{lock}^{L_2} : \textbf{lock}^{L_1 \cup L_2}$.

$$\frac{E[\texttt{vars}](x) = (t, p)}{E; S \vdash x : t', p \rightsquigarrow S', C}\ (\text{Var})$$

where

$$t' = \begin{cases} \textbf{actor}^{\alpha} & \text{, if } t = \textbf{actor} \text{ and } S[\texttt{actors}](x)[\texttt{id}] = \alpha \\ t & \text{, otherwise} \end{cases}$$

**Figure 4.2:** Paragon Type System - Variable Access

For variable access, shown in figure 4.2, we simply look up the type and policy of the requested variable in the environment. However, if the variable holds an actor, we will want to know *which* actor, if possible. Just like for **lock**, we thus let the type **actor** also track which actor it represents.

We have omitted the rule for field access, which mostly follows the same pattern as the variable rule. The main difference is that it may not in general be possible to find which actor identity to look up in the state. If we cannot statically say which actor identity the field access will refer to, a fresh alias will be generated as $\alpha$.

Figure 4.3 shows the rules for expressions that update fields. To type the left-hand side of an assignment, we must ensure that we do not attempt to update a "low" field of a "high" object, or else aliases could be used to leak information. The check that $p_o \sqsubseteq p_f$ reflects this.

$$\frac{\begin{array}{c} E; S \vdash o : T_o, p_o \rightsquigarrow S_o, C_o \\ E[\texttt{types}](T_o)[\texttt{fields}](f) = (t_f, p_f) \\ E; S_o \vdash e : t_e, p_e \rightsquigarrow S_e, C_e \quad t_e :< t_f \end{array}}{E; S \vdash o.f = e : t_f, p_f \rightsquigarrow S', C_f \cup C_e \cup C_{ass}} \text{ (Assignment)}$$

where

$$L = E[\texttt{lockstate}] \rhd S_e[\texttt{lockMods}]$$

$$C_{ass} = \{p_f \sqsubseteq E[\texttt{branchPC}](\star), p_f \sqsubseteq exnPC(S), p_e \sqsubseteq_L p_f, p_o \sqsubseteq p_f\}$$

$$S' = \begin{cases} S_e[\texttt{actors}\ \texttt{scrambled}T_o.f, \\ \qquad \texttt{actors}\{o.f \mapsto (\alpha, T_o.f)\}] & \text{, if } t_e = \textbf{actor}^{\alpha} \\ S_e & \text{, otherwise} \end{cases}$$

**Figure 4.3:** Paragon Type System - Assignments

The test on the types indicates that $t_e$ is assignable to $t_f$, and in particular disregards actor identity annotations. When we move to a system with subtyping, that can be indicated by the semantics of the $:<$ operator.

We can store the identity of $o.f$ in the actor tracker since we know from the language grammar that $o$ must be a simple field access path rooted in a variable. However, such an instance variable will be volatile with stability $T_o.f$.

We omit the rule for assigning to a variable, as it is a simplification of the rule involving instance fields shown here. One thing to be noted though is that $E[\texttt{branchPC}]$ is not defined on $o.f$, since a write to instance field $o.f$ cannot be guaranteed to be "local", hence the use of $E[\texttt{branchPC}](\star)$. For a variable $x$, we would instead use $E[\texttt{branchPC}](x)$ to account for local write effects.

The rules for e.g. post-fix incrementation can also be expressed as special cases of the assignment rule. Similarly, rules for assignment operators are straightforward combinations of the rule for assignment with the rule for binary operators.

One of the arguably most complex rules is that for method calls, as shown in figure 4.4. The first two lines of the premise, containing the typing judgments for the sub-components, are straightforward. The third line simply pattern matches on the method signature stored in the environment.

The constraint on the fourth line checks that if method $m$ *expects* some locks to be open, those locks are indeed guaranteed to be open.

The policy inference constraints for a method call are plenty: those yielded by checking the parent ($C_e$) and argument expressions ($C_i$); for each argument, a check that the argument's policy is no more restrictive than the

$$\frac{\begin{array}{c} E; S \vdash e : T_e, p_e \rightsquigarrow S_0, C_e \\ E; S_{i-1} \vdash e_i : t_i, p_i \rightsquigarrow S_i, C_i \\ E[\texttt{types}](T_e)[\texttt{methods}](m(t_1, \ldots, t_n)) = (t, p, \vec{p}_a, p_w, L, M, Exns) \\ E[\texttt{lockstate}] \rhd S_n[\texttt{lockMods}] \supseteq L \end{array}}{E; S \vdash e.m(e_1, \ldots, e_n) : t, p \rightsquigarrow S'', C_e \cup C_{args} \cup C} \text{ (Call)}$$

where

$$C_{args} = \bigcup \{C_i, p_i \sqsubseteq \vec{p}_a[i]\}$$

$$\begin{aligned} C = {} & E[\texttt{branchPC}] \sqsubseteq p_w, exnPC(S) \sqsubseteq p_w\} \\ & \cup \{Exns(X)[\texttt{write}] \sqsubseteq E[\texttt{exns}](X)[\texttt{write}] \mid X \in \mathrm{dom}(Exns)\} \\ & \cup \{E[\texttt{exns}](X)[\texttt{read}] \sqsubseteq Exns(X)[\texttt{read}] \mid X \in \mathrm{dom}(Exns)\} \\ S' = {} & S_n[\texttt{exns}\{X \mapsto (S_X, Exns(X)[\texttt{write}])\}] \text{ if } X \in \mathrm{dom}(Exns) \\ S_X = {} & S_n[\texttt{lockMods} \rhd= Exns(X)[\texttt{lockMods}], \texttt{actors scrambled}] \\ S'' = {} & S'[\texttt{lockMods} \rhd= M, \texttt{actors scrambled}] \end{aligned}$$

**Figure 4.4:** Paragon Type System - Method Call

declared policy of the parameter ($C_{args}$); checks that the write effect policy of $m$ is no less restrictive than what is allowed by the current branch and exception PCs; and for each exception declared thrown by $m$, a check that its read and write policy modifiers are consistent with those expected by its enclosing handler (i.e. either a **catch** clause or a method signature).

The outgoing state $S''$ will be the final state after checking all arguments, but modified in three ways. First, each exception that may have been thrown by $m$ should now constrain all subsequent expressions and statements until it has been handled, which is reflected by the definition of $S'$. Each $S_X$ will be an approximation of the state at the time $X$ was thrown inside $m$, and will be used when we reach the handler for $X$. Second, we need to update the lock modifiers to reflect any modifications done by $m$. Finally, our alias tracking for actors must account for the fact that all volatile actors could have been changed by executing $m$, so the actors must be scrambled as discussed above. The same goes for the $S_X$ states, since the volatile actors could have been changed before $X$ was thrown.

The rule for top-level method calls with no parent expression is very similar – the only non-trivial change is to use $E[\texttt{methods}]$ instead of $E[\texttt{types}](T_e)[\texttt{methods}]$.

$$\frac{\begin{array}{c} E; S_{i-1} \vdash e_i : t_i, p_i \rightsquigarrow S_i, C_i \\ E[\texttt{types}](T)[\texttt{constrs}](t_1, \ldots, t_n) = (\vec{p}_a, p_w, L, M, Exns) \\ E[\texttt{lockstate}] \rhd S_n[\texttt{lockMods}] \supseteq L \end{array}}{E; S_0 \vdash \textbf{new}\ T(e_1, \ldots, e_n) : T, \bot \rightsquigarrow S'', C_{args} \cup C} \text{ (New)}$$

where

$$C_{args} = \bigcup \{C_i, p_i \sqsubseteq \vec{p}_a[i]\}$$

$$\begin{aligned} C = \{&E[\texttt{branchPC}] \sqsubseteq p_w, exnPC(S) \sqsubseteq p_w\} \\ &\cup \{Exns(X)[\texttt{write}] \sqsubseteq E[\texttt{exns}](X)[\texttt{write}] \mid X \in \mathrm{dom}(Exns)\} \\ &\cup \{E[\texttt{exns}](X)[\texttt{read}] \sqsubseteq Exns(X)[\texttt{read}] \mid X \in \mathrm{dom}(Exns)\} \end{aligned}$$

$$S' = S_n[\texttt{exns}\{X \mapsto (S_X, Exns(X)[\texttt{write}])\}] \text{ if } X \in \mathrm{dom}(Exns)$$

$$S_X = S_n[\texttt{lockMods} \rhd\!= Exns(X)[\texttt{lockMods}], \texttt{actors scrambled}]$$

$$S'' = S'[\texttt{lockMods} \rhd\!= M, \texttt{actors scrambled}]$$

**Figure 4.5:** Paragon Type System - Instance Creation

The rule for instance creation in figure 4.5 is unsurprisingly very similar to that for method invocation found in figure 4.4. The only significant difference is that constructors cannot specify a return policy, as the return policy of a constructor invocation is always $\bot$. This is because a newly created object does not carry any information that is not stored in its fields. Just knowing that the object *exists* carries no information at all.

$$\frac{\begin{array}{c} E; S \vdash e_c : t_c, p_c \rightsquigarrow S_c, C_c \qquad t_c :< \textbf{boolean} \\ E'; S'_c \vdash e_1 : t, p_1 \rightsquigarrow S_1, C_1 \qquad E'; S_c \vdash e_2 : t, p_2 \rightsquigarrow S_2, C_2 \end{array}}{E; S \vdash e_c ? e_1 : e_2 : t, p_c \sqcup p_1 \sqcup p_2 \rightsquigarrow S_1 \diamond S_2, C_c \cup C_1 \cup C_2} \text{ (Cond)}$$

where

$$E' = E[\texttt{branchPC} \sqcup\!= \texttt{p}_\texttt{c}]$$

$$S'_c = \begin{cases} S'[\texttt{lockMods} \rhd\!= L] & \text{, if } t_c = \textbf{lock}^L \\ S' & \text{, otherwise} \end{cases}$$

**Figure 4.6:** Paragon Type System - Conditional Operator

The next expression of interest is Java's ternary conditional operator, for which the typing rule can be found in figure 4.6. Two things are of interest from the perspective of Paragon. The first is that the resulting state is the common denominator of the states resulting from checking the two branches, so we need to merge the outgong states of the branches. The second thing to

note is that if the condition we branch on is of type $\texttt{lock}^L$, we may assume the locks denoted by $L$ to be open when checking the first branch.

$$\frac{E[\texttt{vars}](a_i) = (\texttt{actor}, p_i)}{E; S \vdash_L L(a_1, \ldots, a_n) : p \sqcup \bigsqcup p_i} \text{ (Pred)}$$

$$\frac{E[\texttt{vars}](a) = (\texttt{actor}, p_a) \quad E; S_{i-1} \vdash_L L_i : p_i}{E; S \vdash_C a{:}L_1, \ldots, L_n : p_e \sqcup \bigsqcup p_i} \text{ (Clause)}$$

$$\frac{E; S \vdash_C c_i : p_i}{E; S \vdash \{c_1; \ldots; c_n\} : \textbf{policy}, \bigsqcup p_i \rightsquigarrow S, \emptyset} \text{ (Policy)}$$

**Figure 4.7:** Paragon Type System - Policy Expressions

Finally we turn our attention to policy expressions, presented in figure 4.7. In our simplified type system we assume all policy expressions are built from only simple components. Clauses and lock predicates are not expressions so they each have their own typing judgments. There should be no surprises in any of the rules.

As noted before, combining policies using $\sqcap$ and $\sqcup$ follows the same rules as for normal binary operators (figure 4.1).

### 4.4.3 Typing Statements

The typing judgment for statements is similar to that for expressions – the only difference is that it contains no $\tau$ or $p$ components.

$$E; S \vdash s \rightsquigarrow S', C$$

We overload the $\vdash$ notation since there should be no risk for confusion.

$$\frac{}{E; S \vdash ; \rightsquigarrow S, \emptyset} \text{ (Empty)} \qquad \frac{E; S \vdash s_e : t, p \rightsquigarrow S', C}{E; S \vdash s_e; \rightsquigarrow S', C} \text{ (ExpStmt)}$$

**Figure 4.8:** Paragon Type System - Simple Statements

Figure 4.8 shows the basic rules for empty statements and statements that consist only of expressions.

The rule for conditional statements, presented in figure 4.9, closely resembles the rule for conditional expressions, in figure 4.6. In fact the only difference is the removal of types and policies when expressions changed to statements. Here $t_c :< \textbf{boolean}$ indicates that $t_c$ must be a type assignable to **boolean**, which we note in particular includes the type **lock**.

$$\frac{E; S \vdash e_c : t_c, p_c \rightsquigarrow S_c, C_c \quad t_c :< \textbf{boolean} \quad E'; S'_c \vdash s_1 \rightsquigarrow S_1, C_1 \quad E'; S_c \vdash s_2 \rightsquigarrow S_2, C_2}{E; S \vdash \textbf{if } (e_c)\ s_1 \textbf{ else } s_2 \rightsquigarrow S_1 \diamond S_2, C_c \cup C_1 \cup C_2} \text{ (If)}$$

where $E' = E[\texttt{branchPC} \sqcup\!= \texttt{p}_\texttt{c}]$

$$S'_c = \begin{cases} S'[\texttt{lockMods} \rhd\!= L] & \text{, if } t_c = \textbf{lock}^L \\ S' & \text{, otherwise} \end{cases}$$

**Figure 4.9:** Paragon Type System – Conditional Statement

The rule for **while**-loops, presented in figure 4.10, is arguably the most complex rule we present here. We first note that the branch PC is constrained by the policy of the conditional expression, as expected. Here we also see an example of handling local write effects – any write effects incurred by `break` or `continue` is local to the loop, and need not care about any enclosing branch contexts that the loop appears in.

Second, we note that **while** treats conditional expressions of type **lock** the same way that conditional statements do.

The remaining complexity comes from the multiple entry and exit points of the loop. While-loops (in fact all loops in Java) have three different points of entry depending on how the beginning of the loop was reached. One is the normal entry, where we are poised to evaluate the conditional expression for the first time. The state at that time will be $S$. The second is when execution of the body of the loop completes normally, and we are poised to re-evaluate the conditional expression for a subsequent iteration. The state at that time will be $S_s$. The third possibility is that execution of the loop body completed prematurely through the use of **continue**, and that we are again poised to re-evaluate the conditional expression. The state when **continue** was used in the body will be approximated by the mapping of **continue** in the exception tracker of the state. The definition of $S^*$ reflects these three possibilities.

In order to safely approximate the starting state, we need to first compute the outgoing state of typing $e$ and $s$. $S^*$ combines this outgoing state with the state at the point when we first reached the loop, and is thus a suitably pessimistic approximation of the starting state.

The area of influence (see section 4.3.6) of **continue** extends to the end of the loop, but not to the next iteration. Thus when typing the next iteration we need to remove the influence of **continue** on the exception PC. This is reflected by the definition of $S^{**}$, which is the state we will use as the starting

$$\frac{\begin{array}{c} E;S \vdash e : t_e, p_e \rightsquigarrow S_e, C_e \quad t_e :< \textbf{boolean} \\ E';S_e^* \vdash s \rightsquigarrow S_s, C_s \\ E';S^{**} \vdash e : t_e, p_e \rightsquigarrow S'_e, C'_e \\ E';S_e^{**} \vdash s \rightsquigarrow S'_s, C'_s \end{array}}{E;S \vdash \textbf{while}\ (e)\ s \rightsquigarrow S'', C'_e \cup C'_s} \text{ (While)}$$

where

$$E' = E[\texttt{branchPC} \sqcup\!= p_e, \texttt{branchPC}[\textbf{continue} \mapsto \bot, \textbf{break} \mapsto \bot]]$$

$$S_e^* = \begin{cases} S_e[\texttt{lockMods} \rhd\!= L] & \text{, if } t_e = \texttt{lock}^L \\ S_e & \text{, otherwise} \end{cases}$$

$$S^* = S \diamond S_s \diamond S_s[\texttt{exns}](\textbf{continue})[\texttt{state}]$$

$$S^{**} = S^*[\texttt{exns} /\!= \textbf{continue}]$$

$$S_e^{**} = \begin{cases} S'_e[\texttt{lockMods} \rhd\!= L] & \text{, if } t_e = \texttt{lock}^L \\ S'_e & \text{, otherwise} \end{cases}$$

$$S' = S'_e \diamond S_s[\texttt{exns}](\textbf{break})[\texttt{state}]$$

$$S'' = S'[\texttt{exns} /\!= \textbf{break}]$$

**Figure 4.10:** Paragon Type System - While Loops

state for the second iteration. We use $S[\texttt{exns} /\!= \textbf{continue}]$ to denote the state whose `exns` field does not contain a mapping for **continue**, but which otherwise acts as $S$.

There are also two possible points of (normal) exit of the loop. The first is when the conditional expression evaluates to **false**, in which case the state will be (safely approximated by) $S'_e$. The second is if the body of the loop ends prematurely through the use of **break**, in which case the state will be reflected by the mapping of **break** in the exception tracker of the state, analogous to **continue**. The definition of $S'$ reflects these two possibilities.

In the actual result state we must also cancel the area of influence of **break**. The definition of $S''$ captures this.

Finally, since the state we use on the second pass is the safe approximation, the constraints we are interested in collecting are the ones incurred by the second checks of $e$ and $s$, i.e. $C'_e$ and $C'_s$.

Figure 4.11 shows the type rules for goto-like constructs. In the rule for **continue**, we let the current effective PC act as constraint on subsequent statements by adding it to the exception PC. This will be in effect up until the "catch point" for **continue** – i.e. until the end of the nearest enclosing

$$\frac{}{E; S \vdash \textbf{continue};\ \rightsquigarrow S', \emptyset}\ \text{(Continue)}$$

where $S' = S[\texttt{exns}\{\textbf{continue} \mapsto (S, w)\}]$

$w = E[\texttt{branchPC}](\textbf{continue}) \sqcup exnPC(S)$

$$\frac{E; S \vdash e : t, p \rightsquigarrow S_e, C_e \quad E[\texttt{return}] = (t_r, p_r) \quad t = t_r}{E; S \vdash \textbf{return}\ e;\ \rightsquigarrow S', C_e \cup C_r}\ \text{(Return)}$$

where

$S' = S_e[\texttt{exns}\{\textbf{return} \mapsto (S_e, p_r)\}]$

$C_r = \{E[\texttt{branchPC}](\textbf{return}) \sqsubseteq p_r, exnPC(S_e) \sqsubseteq p_r, p \sqsubseteq_L p_r\}$

$L = E[\texttt{lockstate}] \rhd S_e[\texttt{lockMods}]$

**Figure 4.11:** Paragon Type System - Exception-like Statements

loop, as seen by the rule for **while** in figure 4.10. We also store the current state, which will be the state at the beginning of the next iteration of the loop.

The rule for **break** is identical to the rule for **continue**, only changing the keyword used, so we omit it.

The rule for **return** $e$ is also similar, but does not use the current PC for the combined write effect and exception PC contribution. Instead we use the current method's return policy, declared as a modifier on the method and stored in the environment when checking the method's body. The return policy acts as the write effect of the **return** statement, and must be checked against the relevant branch and exception PCs. It is also the policy given to the value returned, and thus it must be valid to re-annotate the return expression to this policy in the current lock state, which is checked by the test $p \sqsubseteq_L p_r$.

We omit the rule for **return** statements without an accompanying expression, i.e. returning from a method with return type **void**. The rule is nearly identical to that for **continue**. The only difference apart from the keyword is that the rule for **return** has the test $E[\texttt{return}] = (\textbf{void}, \top)$ in the premise.

The rule for **throw**, in figure 4.12, is very similar to that for **return**. The key difference is that for exceptions we allow the write effect of the exception to be different from the policy on the value thrown. This will not lead to unintended leaks since the conceptual assignment of the value thrown to the parameter of the catch clause is a local write inside the exception's area of

$$\frac{E; S \vdash e : X, p \rightsquigarrow S_e, C_e}{E; S \vdash \textbf{throw}\ e;\ \rightsquigarrow S', C_e \cup C}\ \text{(Throw)}$$

where

$$\begin{aligned} w_X &= E[\texttt{exns}](X)[\texttt{write}] \\ S' &= S_e[\texttt{exns}\{X \mapsto (S, w_X)\}] \\ C &= \{\ E[\texttt{branchPC}](X) \sqsubseteq w_X, exnPC(S_e) \sqsubseteq w_X, \\ &\qquad p_e \sqsubseteq_L E[\texttt{exns}](X)[\texttt{read}]\ \} \\ L &= E[\texttt{lockstate}] \rhd S_e[\texttt{lockMods}] \end{aligned}$$

**Figure 4.12:** Paragon Type System - Exceptions

influence.

For catching exceptions, we first make the simplifying assumption that all **try**-**catch**- **finally** statements are unrolled into nested **try**-**catch** and **try** - **finally** , each with just a single catch- or finally-block. The rules for these two statements are found in figure 4.13.

For **try**-**catch**, the first thing we must do is to register the handler in the environment used when checking the try-block. This means storing the read and write policies associated with the caught exception type. The read policy is declared as a modifier on the parameter to the catch block. The write policy is not declared anywhere, and so we must infer it from the context we have available. We thus introduce a fresh policy variable, which will then appear in a number of constraints in both $C_t$ and $C_c$.

The remaining complexity only amounts to ensuring that the proper states and environments are used in the right places, similar to rules we have already discussed.

One thing to note though is the appeal to a function *readPol*. This function hides quite a bit of complexity which we will not cover here. Loosely, it picks out the read (or return) policy modifier from the set of modifiers $ms$, and evaluates this policy statically. In doing so it will use, among other things, $E[\texttt{typemethods}]$ and the actor identities found in $S[\texttt{actors}]$, which is why both $E$ and $S$ are needed to provide the context for *readPol*. We will see more examples of such extraction functions in subsequent rules, but will not cover their definitions.

For **try**- **finally** , we know that regardless of how the execution of the try-block ends, the finally-block *will* be executed. Thus, for any exception (or goto-like statement) thrown *inside* the try-block, the finally-block is not part of that exception's area of influence. The exception PC at the start of

$$\frac{E'; S \vdash b_t \rightsquigarrow S_t, C_t \quad E^*; S^* \vdash b_c \rightsquigarrow S_c, C_c}{E; S \vdash \textbf{try}\ b_t\ \textbf{catch}\ (ms\ T\ x)\ b_c \rightsquigarrow S', C_t \cup C_c}\ \text{(TryCatch)}$$

where
$$p_r = E; S \vdash readPol(ms)$$
$$E' = E[\texttt{exns}\{T \mapsto (\texttt{read} = p_r, \texttt{write} = \pi)\},\ \texttt{branchPC}\{T \mapsto \bot\}] \text{ where } \pi \text{ is fresh}$$
$$E^* = E[\texttt{vars}\{x \mapsto (T, p_r)\}]$$
$$S^* = S_t \diamond S_t[\texttt{exns}](X)[\texttt{state}]$$
$$S' = S_c[\texttt{exns} /= T]$$

$$\frac{E; S \vdash b_t \rightsquigarrow S_t, C_t \quad E; S^{**} \vdash b_f \rightsquigarrow S_f, C_f}{E; S \vdash \textbf{try}\ b_t\ \ \textbf{finally}\ \ b_f \rightsquigarrow S', C_t \cup C_f}\ \text{(TryCatch)}$$

where
$$S^* = S_t \diamond \diamond\{\ S_X \mid X \in \text{dom}(S_t[\texttt{exns}], S_t[\texttt{exns}](X)[\texttt{state}] = S_X\ \}$$
$$S^{**} = S^*[\texttt{exns} = S[\texttt{exns}]]$$
$$S' = S_f \diamond S_f[\texttt{exns} = S^*[\texttt{exns}]]$$

**Figure 4.13:** Paragon Type System - Try-Catch-Finally

the finally-block is thus the same as at the start of the try-block, which the definition of $S^{**}$ captures. However, any statements *after* the finally-block are part of the area of influence of all exceptions, including those thrown inside the try-block, so the resulting state must reinstate the effect of those on the exception PC for subsequent computation.

Finally we turn to the statements added by Paragon, namely those that handle lockstate modifications. The type rule for the **open** statement is fairly straightforward. All the arguments must be actors, and the result of the statement is to register in the state that the lock has been opened for the identities that those actors represent. Opening the lock causes a potentially visible state change, so the policy of the lock must not be lower than the current PC.

The only potentially non-obvious part of the rule is the test that the policies of each actor argument must not be more restrictive than the policy on the lock, i.e the test $p_i \sqsubseteq p_L$. This serves the same purpose as the restriction on field updates, in figure 4.3, that the policy on the field must not be less restrictive than the policy on the object the field belongs to – we may not update "low" parts of the state for a "high" object. Opening a lock changes the (lock)state associated with its actor arguments, so the policy on the lock family must similarly not be too liberal.

$$\frac{E; S_{i-1} \vdash e_i : \textbf{actor}^{\alpha_i}, p_i \rightsquigarrow S_i, C_i \quad E[\texttt{locks}](L) = (n, p_L)}{E; S_0 \vdash \textbf{open}\ L(e_1, \ldots, e_n);\ \rightsquigarrow S', C} \text{ (Open)}$$

where

$$S' = S_n[\texttt{lockMods} \rhd\!= (\emptyset, \{L(\alpha_1, \ldots, \alpha_n)\})]$$

$$C = \{E[\texttt{branchPC}](L) \sqsubseteq p_L, exnPC(S) \sqsubseteq p_L\} \cup \bigcup\{C_i, p_i \sqsubseteq p_L\}$$

$$\frac{E; S_{i-1} \vdash e_i : \textbf{actor}^{\alpha_i}, p_i \rightsquigarrow S_i, C_i \quad E[\texttt{locks}](L) = (n, p_L) \quad E'; S_n \vdash b \rightsquigarrow S_b, C_b}{E; S_0 \vdash \textbf{open}\ L(e_1, \ldots, e_n) b \rightsquigarrow S_b, C_b \cup C} \text{ (OpenIn)}$$

where

$$E' = E[\texttt{lockstate} \cup\!= \{L(\alpha_1, \ldots, \alpha_n)\}]$$

$$C = \bigcup\{C_i, p_i \sqsubseteq p_L\}$$

**Figure 4.14:** Paragon Type System - Lockstate Modification Statements

The rule for **close**, omitted here, is nearly identical to the rule for open – the only change needed is to swap the tuple to $(\{L(\alpha_1, \ldots, \alpha_n)\}, \emptyset)$ when registering the modification in the state.

The rule for the scoped version of **open** is interesting in two regards. First, since the opening of the lock has a block scope, we use the `lockstate` field of $E$ to register it as open for the extent of the enclosed block. This naturally also means that the status of the lock will be the same when leaving the block as it was when the block started.

What the rule does not currently capture though is the restriction that the enclosed block may not close the lock. To handle that restriction we would need to register in the environment what locks are currently untouchable, and check against that when closing locks. It is a trivial fix, but we leave it out since it adds unnecessary complexity to the presentation.

The second interesting thing to note is that there is no need to check the modification of the lock against the PC. Since the status of the lock will be the same in the outgoing state as it was in the incoming, the modification can never be visible outside the scope of the statement itself, so the incurred write effect is purely local.

### 4.4.4 Typing Blocks and Block Statements

$$\frac{}{E; S \vdash \{\} \rightsquigarrow S, \emptyset} \text{ (EmptyBlock)}$$

$$\frac{E; S \vdash s \rightsquigarrow S_s, C_s \quad E; S_s \vdash \{ss\} \rightsquigarrow S'', C}{E; S \vdash \{s\ ss\} \rightsquigarrow S', C_s \cup C} \text{ (BlockStatement)}$$

$$\frac{\begin{array}{c} E; S \vdash e : t_e, p_e \rightsquigarrow S_e, C_e \quad t_e :< t \\ E'; S'_e \vdash \{ss\} \rightsquigarrow S', C \end{array}}{E; S \vdash \{ms\ t\ x = e;\ ss\} \rightsquigarrow S', C_s \cup C} \text{ (LocalVarDecl)}$$

where

$$r = E; S \vdash readPol(ms)$$
$$E' = E[\texttt{vars}\{x \mapsto (t, r)\}]$$
$$S'_e = \begin{cases} S_e[\texttt{actors}\{x \mapsto \alpha\}] & \text{, if } t_e = \textbf{actor}^{\alpha} \\ S_e & \text{, otherwise} \end{cases}$$

**Figure 4.15:** Paragon Type System - Blocks and Block Statements

We overload the $\vdash$ notation yet again for the typing judgment for blocks in figure 4.15. The rules add little novelty, and differ from previous rules only in that the environment is updated for subsequent iterations. The rule for local variable declarations with initialisers is unsurprisingly very similar to the rule for assignments.

We have omitted the rule for declarations without initialisers since it is mostly a simplification of the rule given. However, there is one interesting difference – if $t$ is **actor** and no initialiser is given, then $\alpha$ will be $\texttt{fresh}\, k$ for a generated $k$.

We also omit the rule for local lock family declarations, since it is largely trivial, only registering the policy, arity and properties of the lock family in the environment. Similarly we have not included the special case for policy declarations. These are not quite as trivial in that they require static evaluation of the initialiser expression, but we do not go into the details of that here.

### 4.4.5 Typing Method Declarations

Finally we can look at how to type a complete method declaration, show in figure 4.16. The rule itself is not so complicated, but contains several things that need explanations.

$$\frac{\begin{array}{c} E'; S' \vdash body \rightsquigarrow S'', C \quad \models C \\ M \sqsupseteq S''[\texttt{lockMods}] \\ MX_i \sqsupseteq S''[\texttt{exns}](X_i)[\texttt{state}][\texttt{lockMods}] \end{array}}{\begin{array}{l} E; S \vdash ms\ t\ m(ms_1\ t_1\ x_1, \ldots, ms_n\ t_n\ x_n) \\ \qquad \textbf{throws}\ mx_1\ X_1, \ldots, mx_k\ X_k \quad body \end{array}} \text{(MethodDecl)}$$

where

$$\begin{aligned} r &= E; S \vdash readPol(ms) \\ w &= E; S \vdash writePol(ms) \\ p_i &= E; S \vdash readPol(ms_i) \\ M &= E; S \vdash lockMods(ms) \\ L &= E[\texttt{lockProps}] \cup E; S \vdash expects(ms) \\ rx_i &= E; S \vdash readPol(mx_i) \\ wx_i &= E; S \vdash writePol(mx_i) \\ MX_i &= E; S \vdash lockMods(mx_i) \\ E' &= E[\texttt{return} = (t, r), \texttt{lockstate} = L, \texttt{vars}\{x_i \mapsto (t_i, p_i)\}, \\ &\qquad \texttt{exns}\{X_i \mapsto (rx_i, wx_i)\}, \texttt{branchPC}\{\star \mapsto w, x_i \mapsto \bot\}] \\ S' &= S[\texttt{lockMods} = (\emptyset, \emptyset), \texttt{exns} = \{\}, \\ &\qquad \texttt{actors}\{x_i \mapsto (\texttt{alias}\ k_i, \texttt{stable})\}] \\ &\qquad \forall i. t_i = \textbf{actor}, k_i\ fresh \end{aligned}$$

**Figure 4.16:** Paragon Type System - Method Declarations

To type the body we must first set up the proper environment, which constitutes a large portion of the complexity in this rule. We use auxiliary extraction functions to find and evaluate the various modifiers for the method itself, its parameters, and the thrown exceptions. One thing to note is that we let the starting lockstate include all the lock properties[5], similar to what we did in section 3.7.

Most of the setup of $E'$ and $S'$ should be self-evident. We note that the `branchPC` is set up to treat assignments to the parameter variables as local to the method, while uncaught exceptions cause write effects that escape the scope of the method body. The actor tracking must generate a fresh, aliased identity for each parameter of type **actor**. Since parameters are local variables, their identities are stable in the face of method calls.

[5]... glossing over the fact that the properties are clauses and not locks. However, if we treat each lock as a clause with that lock in the head and an empty predicate list, the union is well-typed.

Once the body has been typed, we need to ensure two things. First we must solve the constraints gathered from the body, as indicated by $\models C$. We have not yet investigated the details of constraint solving, but note that our constraints are fairly simple inequalities involving literal policies, inferrable policy variables (from local variables with no specified policy), rigid policy variables (from policy-polymorphic parameters, which cannot be constrained in any way), joins and meets. We expect to find standard methods to solve such constraints. We further note that in the worst case, we can ask the programmer to put in some extra policy annotations to guide the constraint solver.

The second thing we must ensure is that the declared modifications on the lockstate, both for normal return and for exceptions, are safe approximations of the modifications the body will actually perform. We define a partial order $\sqsubseteq$ on lock modifications as follows:

$$(C_1, O_1) \sqsubseteq (C_2, O_2) \text{ iff } C_1 \subset C_2 \wedge O_1 \supset O_2$$

Thus, if $M_1 \sqsubseteq M_2$ we know that $M_2$ closes at least the locks that $M_1$ does, and opens only locks that $M_1$ also opens. The two constraints in the rule will then ensure that the signature of the method is a safe approximation concerning what the lockstate will be after the method has been applied.

We have not covered top-level field or lock family declarations, but note that they are very similar to their local counterparts.

## 4.5 Compiling Paragon

In the previous sections we have presented the front-end of the language Paragon: its features and expected behavior, as well as the static semantics. In this section we briefly discuss how to compile a Paragon program into vanilla Java, and how the runtime aspects of Paragon are represented.

Once we know that a given program satisfies the intended information flow properties, we can safely remove all type-level aspects of policies, locks and actors. We must still retain the runtime aspects, and in some cases demote from the former to the latter.

All actor and policy type parameters on methods are demoted to formal (value) parameters, and type arguments to method calls are demoted to normal arguments. For type parameters to classes, each type parameter is also added as a field to the class, and as a formal parameter to each constructor of the class, with an initialisation at the start of the constructor.

Uses of **instanceof** that check against a type with actor or policy parameters are extended to compare against the fields representing those parameters. For example, `x instanceof Vector<a,int>` where the class is defined `Vector<actor A,T>`, is compiled to:

```
x.A == a & x instanceof Vector<int>
```

Runtime casts to types with parameters are affected similarly.

Actors need only one property at runtime – to be uniquely distinguishable from each other. Many different representations could be considered – our current prototype implementation simply uses instances of `Object` to represent actors, and distinguishes between them via hash codes. All declarations of actor fields and variables with no accompanying initialisation are given initialisers that generate a unique actor representation – which in our prototype simply means **new** `Object()`.

Lock families need to support opening and closing of individual locks in the family, parameterised by actors, as well as querying of current status. Again many different representations could be considered. Our prototype uses a `java.util.Set`, whose entries are arrays of actor representations for the actors for which a lock in the family is open. The three operations are then obvious. As a special case, locks with no parameters are simply boolean variables. This model is not sophisticated enough to handle lock *properties* – hence our current prototype does not take properties into account at runtime.

The only Paragon-specific statements are **open** and **close**, which become manipulations of the lock representations discussed above. The scoped versions need a bit more effort, as they need to ensure that they leave the lock in the same state as it was at the start of the block, even in the presence of exceptions.

Policies are quite intricate and need to support a number of operations at runtime, in particular meets, joins and entailment. Our current prototype does not yet support runtime policies, and the prototype type checker does not let programs through that rely on runtime aspects of policies. This includes in particular code that uses runtime tests of policy entailment – such code is currently unsupported.

## 4.6 A Comparison with Jif

Comparing Paragon to Jif is inevitable. Jif stands out as the only existing information-flow-typed programming language to date, and is at the same time a competitor and a source of inspiration. Due to the unique position Jif has enjoyed in the domain of information flow research over many years,

much research has been done using Jif and DLM for context and examples. It is thus natural to ask how research done on or with Jif can carry over to Paragon.

In this section we make a brief but detailed competitive comparison of Paragon and Jif. We begin by giving an overview of Jif and where it differs from the DLM that we have discussed in previous sections. Second we discuss the perceived short-comings of Jif, and how we deal with those aspects in Paragon. We then go on to compare various language features, and point out where Jif, or Jif-related research, has directly influenced our design choices. Finally we show how Paragon can encode the DLM as a library, arguing, as of yet without any hands-on experience, that Paragon could be used as a drop-in replacement for Jif in existing examples.

### 4.6.1 The Jif Language

Jif, (and its predecessor JFlow, [Mye99, MZZ+06]) is a version of Java which adds statically-checked information-flow annotations in the form of DLM labels. Jif extends the core DLM model with a number of important features, including:

- Authority and Selective Downgrading: any piece of code in a Jif program runs on behalf of a certain set of principals, known as the *authority* of the code. The language contains a *declassify* operator which allows the policy of an expression to be weakened. But not just any weakening is permitted. Only parts of the policy owned by the current authority may be weakened in this way. For example suppose a piece of data is labeled with the policy $\{\text{Alice} : \text{Chuck}, \text{Dave} \ ; \ \text{Bob} : \text{Dave}, \text{Eve}\}$ as above. If the code runs with at least the authority of Alice then it can be declassified to $\{\text{Alice} : \text{Chuck}, \text{Dave}, \text{Eve} \ ; \ \text{Bob} : \text{Dave}, \text{Eve}\}$ in which case the information may then flow to Eve.

- Robustness: Jif (since version 3) can optionally be run in "robust" mode [ZM01, MSZ04]. In robust mode the decision to declassify and the data to be declassified cannot be influenced by low-integrity data.

Many other features of Jif are purely programming language issues rather orthogonal to the DLM and policies, and concern the tracking of information flows and the way the type system expresses these. Examples of such features include the treatment of exceptions, and the support for principals and labels that are only known at runtime. We discuss such features in 4.6.3.

### 4.6.2 Jif Concerns

We had two main concerns with Jif when starting the Paragon project.

Firstly, and most severe, there is no complete semantic security model for either Jif or the DLM, meaning Jif only provides *intuitive* guarantees that a well-typed program is actually secure.

In contrast, Paralocks has a formal semantic security model. In section 3.5 we proved, for a type system for a simple imperative language, that well-typed programs are secure in that model. The full Paragon type system is a far larger beast, and we have not yet attempted to prove a corresponding property. It remains as one of the most important items on the list for future work. We surmise that the situation is still better than for Jif though, where only selected isolated fragments, namely robust declassification [MSZ04], runtime principals [TZ04], and runtime labels [ZM07b], have been treated formally – and even then in languages far from Java.

Second, and perhaps more subjective, we felt that the policy model in Jif was too restrictive, in that it could not be used to express many of the proposed idioms for programming with information flow control. As case in point, the original DLM model could not express robust declassification. To mitigate this short-coming, Jif has *added* integrity labels on top of the already existing confidentiality labels, as integrity aspects could not be expressed using already existing features.

In contrast, flexibility has been one of the main design goals for our work on Paralocks and Paragon. We have shown in section 4.2 that Paragon can encode a number of existing idioms for information flow policies, including robust declassification, which we believe serves as strong evidence that we have succeeded.

### 4.6.3 Feature Comparison

**Types and policies vs labeled types** Jif labels can only express the DLM notions of confidentiality and integrity, and requires both aspects to be specified. In contrast, Paragon policies can express a wide range of information flow idioms, and can include confidentiality and integrity aspects as needed, as shown in the examples in section 4.2.

In Jif, every value has a *labeled type*, bundling types and labels together both syntactically and semantically. We find this unfortunate since, while both types and labels affect type checking, they are largely orthogonal concepts as far as a programmer is concerned.

Paragon keeps policies and types separate, keeping the former specified through modifiers instead of annotations directly on the types. We feel this

allows for a cleaner separation of the Paragon additions from the vanilla Java code, making Paragon code more accessible to a Java programmer.

**Locks vs authority and delegation** The main strength of Paragon over Jif is the generality of the concept of locks. The Jif notions of authority and delegation are in Paragon just special cases of lock families. Queries to the *acts-for* hierarchy in Jif then simply become a particular kind of runtime lock queries in Paragon. Similarly *method constraints*, used to specify constraints regarding the *acts-for* hierarchy and the authority of the calling code, are just special cases of *expects* lock state modifiers.

Furthermore, Paragon allows dynamic changes to locks during program execution, which when considered in Jif terms means that it would e.g. be possible to encode mechanisms to grow and shrink the acts-for hierarchy dynamically. In Jif, the notions of authority and delegation are static.

**Exception handling** Exceptions are handled largely in a similar fashion in Jif and Paragon, and in many ways Paragon has benefited from Jif serving as a trail-blazer. For instance we did not need to realise ourselves the potential problems of unchecked exceptions, or the need for null pointer analysis.

Regarding the effect of exceptions on the PC, Paragon takes a different approach to reasoning about it, by considering areas of influence instead of PC traces, but the approaches are identical regarding expressive power.

**Static initialisers** Jif identifies the same problems we do for static initialisers, a problem that has also been studied by Nakata and Sabelfeld [NS10]. Jif adopts a more restrictive solution to the problem, by restricting the initialisers not only to not throw exceptions, but also to not have any side-effects. As noted in section 4.3.7, Paragon handles side-effects by requiring them to be declared as a modifier to the class.

**Type parameters** Jif saw the light of day before the introduction of Generics in Java. Still the authors of Jif recognised the need for parameterising classes on labels and principals, so Jif rolled its own form of parametric polymorphism, only allowing Jif-specific parameter kinds. While this work is quite impressive (and as a side-track led to the development of PolyJ [MBL97], a competitor to GJ [BOSW98] that later became Java Generics), we have the luxury of having Java Generics as a starting point. Our type parameter extensions thus syntactically fit more nicely with what is already available in Java.

One particular advantage of this is that we can also build on Java's mechanism for type-parameterised *methods*, and pass arguments intended to affect the method's signature as type arguments instead of formal arguments as in Jif.

Regarding expressive power, we are not aware of any difference between our version and that of Jif, if only actor and policy parameters are considered. Our lock set parameters have no counterpart in Jif.

**Aspects of dependent typing** Both Jif and Paragon have limited forms of dependent types. Labels and principals in Jif, and policies and actors in Paragon, can be used both as first-class values at runtime, and in the specification of other labels/policies.

In Jif, a label or principal may be hoisted from the value level to the type level, and used as a type argument or in the construction of new labels, assuming the expression representing the label or principal is a *final access path*. To make programming with type arguments smoother, Jif introduces the restriction that formal parameters of a method are always final, so can always be used as the root for a final access path e.g. when writing the signature of the method.

The notion of final access paths carries over into Paragon, where they can be used to specify arguments to parameterised types, or in policy annotations. However, for type parameters on *methods* we do not require finality, as the requirements for consistency of types do not apply in the same way. Also, exactly *because* we have type parameters on methods (unlike Jif), the reason to restrict formal parameters to be final partly falls, since we would pass arguments that affect the method's signature as type parameters instead.

**Runtime policies** Runtime policies have been studied by Zheng and Myers in the context of Jif and the DLM [ZM07a], and their work is the basis for the current evolution of runtime labels in Jif. Their ideas can largely be directly applied to Paragon as well, and the language design regarding runtime policies in Paragon is directly taken from their work.

### 4.6.4 Example: Encoding the DLM

In 3.6.2 we showed how Paralocks can be used to encode the DLM, proving Paralocks strictly more general. In this section we show how that encoding can be implemented as a Paragon library, which we surmise can be used as a drop-in replacement for Jif in example code, albeit with a few caveats.

For the *ActsFor* hierarchy, we need a lock family that represents a reflexive, transitive relation on actors, similar to the one in the encoding of lexically scoped flows:

```
public static reflexive transitive
  lock ActsFor(A,actsForA);
```

To mark the authority of any given piece of code we use a unary lock *RunsFor*:

```
public static lock RunsFor(A) {
  RunsFor('x): ActsFor('x,'y), RunsFor('y)
  };
```

Here no syntactic sugar will do to express the interplay between *ActsFor* and *RunsFor*.

Moving one step closer to Jif, we also introduce an explicit declassification function to ensure that declassification only happens at explicitly marked locations. We use a similar trick like in the example with lexically scoped flows to let our `declassify` function take a code block as its argument, inside which the declassification may take place:

```
public static lock Declassify;

public static !W ~L <policy W, lock[] L>
  void declassify(Decl<W,L> block) {
    open Declassify { block.go() } }

public interface Decl<policy W, lock[] L>{
  !W ~L ~Declassify void go(); }
```

Finally we need to represent DLM *labels* in this framework.The encoding of DLM into Paralocks was shown and proven correct in [BS10], and the interested reader is referred there for the details. We can simplify the programming of our encoding by expressing it at the level of a single clause rather than a whole label. The policy of a label can then be represented as the join of those respective clauses. We can thus define the following type method[6]:

[6]Note that the *varargs* parameter, denoted by the ellipsis, is vanilla Java and not a Paragon innovation.

```
public typemethod policy
  lbl(actor owner, actor... readers) {
    policy c =
      { 'x : RunsFor(owner), Declassify };
    for (actor reader : readers) {
      c += { 'y : ActsFor(reader, 'y) }; }
    return c; }
```

We can now form full labels by joining such clauses together, so for instance the DLM label

```
    { o1: r1,r2; o2: r2,r3 }
```

can be written as

```
    lbl(o1,r1,r2) * lbl(o2,r2,r3)
```

Our encoding does not cover the integrity and robustness aspects of Jif, but we surmise that they can be handled in a dual manner, similar to what we did in the example in section 4.2.2.

# Chapter 5

# Related work

In this chapter we look at related work along the three axes we pointed out in the introduction: Policy specification mechanisms (5.1), semantics of information flow (5.2), and programming languages with information flow control capabilities (5.3).

Further, we also look at work on the concept of *typestate* (5.4) which is closely related to locks in Paragon.

## 5.1 Policy Specification Mechanisms

As we have argued previously, most research on information flow security and declassification to date use one of only two policy specification mechanisms: A lattice model following the work by Denning [Den76], or the Decentralised Label Model (DLM) by Myers and Liskov [ML97].

In the lattice model, policies are defined by a lattice $(\mathcal{L}, \sqsubseteq)$, where $l \in \mathcal{L}$ is a *level*, with levels ordered by the partial order relation $\sqsubseteq$. The simplest example of such a lattice is one with only two levels, "high" and "low". We can define it as a lattice with $\mathcal{L} = (H, L)$ where $L \sqsubseteq H$. A significant number of information flow systems uses such a simple model at its core, e.g. Askarov and Sabelfeld's work on gradual release [AS07]. Such systems clearly focus on aspects other than policy specification.

A lattice can consist of explicitly named and ordered levels, as in the "high-low" case, but more common is systems where the lattice levels correspond to e.g. a powerset lattice of principals [Sim03, AB05], or systems that work with any arbitrary lattice [CM04, CM05, ZM07a, Bou05].

The other mechanism, the DLM, we have already discussed at length, but note that it is used as the policy specification mechanism for plenty of research not specifically targeted at Jif, e.g. [KHHJ08, HTHZ05, TZ04].

Paralocks can encode both a lattice model and the DLM, and is thus strictly more general than these models.

There are systems that introduce other policy specification mechanisms. One example is the Rx policy language by Swamy et al [SHTZ06]. This work (and the more recent refinement by Bandhakavi et al [BWW08]) is the only other language-based security work of which we are aware which uses roles in an information-flow setting. The main thrust of their approach is to specify and manage information flows which are caused by policy changes. Role management ideas are used to control policy updates. In common with this approach, our semantics also tracks information flows caused by "role management". We believe that many features of their meta-policies can be directly encoded using Paralocks, but we have yet to investigate such examples.

Nanevski et al [NBG11] introduce *Relational Hoare Type Theory (RHTT)*, a specification language and verification system that can express information flow and access control policies via value-dependent types. The strength of their system over Paralocks is that they can express policies in terms of arbitrary program state, not just the locks and actors that Paralocks allows. This fact also lets their system include the "what" dimension of declassification in a natural way. This strength is also its biggest drawback however - a full value-dependent type system induces a much heavier enforcement machinery, including interactive proofs. This puts far higher requirements on the programmer, something we deliberately want to stay clear of with our work on Paralocks.

There are several policy languages in the access control and authorisation area which have some superficial similarity with Paragon/Paralocks, since they are based on Datalog like clauses to express properties like delegation and roles, see e.g. [Jim01, LMW02, DFK06, BFG07]. Key differences are (i) the information flow semantics that lies at the heart of Paragon, and (ii) the fact that the principal operation in Paragon is comparison and combination of policies, whereas is the aforementioned works the only operation of interest is (run-time) querying of rules.

We note some further similarities to work on RBAC models. Paralocks permit finer granularity than standard RBAC, with the use of user-specific policies. This level of control appears similar to that provided by *role templates* [GI97]. Our ability to model accesses which are triggered by arbitrary state conditions (modeled via locks) has similarities to *environment roles* [CLS+01]. Related to this, the role activation rules of the OASIS model have a superficial similarity with Paralocks policies (see e.g. [BEM03]) although these rules would be more like lock invariant specifications in our model.

The extension to recursive Paralocks described in Section 3.7 brings the work much closer to the logic-based access control work (e.g. [Jim01, DeT02, LMW02]). One line of work by Dougherty et al [DFK06] deals specifically with the issues that arise in situations where changes in the environment entail dynamic changes to access control policy. This is analogous to our problem of reasoning about policies in the presence of a program which has side-effects on the policy.

## 5.2 Semantics of Information Flow

Semantic models for complex information flow policies have historically been problematic. In some cases – e.g. in the DLM – there is simply no information flow model. In others (e.g. Tse and Zdancewic [TZ04]) the semantic models are simply *noninterference in the absence of policy change*. For semantic models of declassification and other dynamic information flow policies which attempt to do more than this (e.g. the "noninterference between policy updates" approach in Swamy et al [SHTZ06, BWW08]), many semantic models suffer from *flow insensitivity*. Flow insensitivity here means that the semantic conditions are not really fully semantic, since they flag insecurity simply because they do not have an sufficiently accurate model of the context of a given "insecure looking" subcomputation.

The notion of flow (in)sensitivity comes from the static analysis world, where it is used to characterise program *analyses*, and is not used to describe the underlying semantic property.

The flow insensitivity problem arising in many approaches [MS04, EP05, EP03, AB05, Dam06, MR07, BCR08, LM08] all come about through somewhat related bisimulation-like definitions. But flow insensitivity can arise, in varying degrees, in other styles of model too. For example, Swamy et al [SHTZ06] deal with a detailed model of information flow policy updating. The semantics is phrased in terms of the trace segments in between policy updates, and asserts noninterference for the programs at the beginning of each of these segments. This is a resetting approach since it reasserts noninterference at intermediate program points, and thus becomes flow insensitive. As another example, flow insensitivity also arises in the definition of qualified robust declassification from Myers et al [MSZ04] which uses a "scrambling" semantics for endorsement (upgrading of integrity) which nondeterministically resets the value of a variable after its endorsement.

As mentioned previously, the knowledge based approach used in this thesis is inspired by the Gradual Release work [AS07]. Similar uses of knowledge sets appear earlier – e.g. Dima et al [DEG06] – and many of the classic non-

interference definitions have a knowledge or "deducibility" flavour. However Askarov and Sabelfeld [AS07] appear to be the first to use this style of definition to reason about the semantics of declassification.

Banerjee et al [BNR08] extend Gradual Release in a rather orthogonal direction, by allowing declassifications to carry a logical specification of *what* is declassified, and under what condition. Their approach is based on *agreement predicates* as pre- and post-conditions to commands. Predicates constrain the memories compared in traces to only start with such pairs of memories that agree on the values of certain expressions, leading to a form of *delimited release* [SM04].

Paralocks does not provide direct control of the "what" dimension of declassification [SS05]. Although certain simple "what" policies are easily encoded in the Paralocks language, our semantic model cannot make any formal guarantees about such examples. Balliu et al [BD11] show how the knowledge-based style can be naturally represented by *epistemic temporal logic*. Their starting point is Gradual Release, which they also extend to handle various dimensions of declassification, including a characterisation of "what" similar to that of Banerjee et al [BNR08]. Unlike Paralocks but like Gradual Release, their model of the "where" dimension only allows successively more liberal policies.

## 5.3 Information Flow Programming Languages

**Languages with explicit information-flow tracking** Two "real-sized" languages stand out as providing information-flow primitives as types. The first is Jif [MZZ+06], which we have already discussed at length in the previous section. A recent extension of Jif is *Fabric* [LGV+09], adding support for distributed programming and transactions. Fabric is a complete, implemented system for writing distributed applications, but shares the same weakness that Jif does in that it does not include any semantic characterisation of security, neither for its extensions or for the core Jif language that it is built on.

The second is FlowCaml [Sim03] which we discussed briefly in the introduction. FlowCaml is a subset of OCaml extended with information flow annotations on types. In FlowCaml policies (annotations) are security levels chosen in a user-definable lattice. The policy is the basic multilevel security, with no support for e.g. declassification policies. On the other hand FlowCaml supports full type inference in the ML tradition, and is built on a semantic formalisation which is rather close to the full language [PS03].

**Compilers performing IF tracking** Information-flow tracking can be performed in a language which has no inherent security policies, lattice-based or otherwise. In such a setting one tracks the way that information flows from e.g. method parameters to outputs. The Spark Examiner, a commercial tool for static analysis and verification for a safety-critical subset of Ada, contains such an analysis [CH04].

Hammer and Snelting [HS09, Ham10] explain how state-of-the-art program slicing methods can support a more accurate analysis of such information flows in Java (e.g. both flow sensitive and object sensitive).

**Encoding Information Flow Policies with Expressive Type Systems** With suitably expressive type systems and abstraction mechanisms, static information flow constraints can be expressed via a library. Li and Zdancewic [LZ10] showed how to provide information-flow security also as a library. Tsai et al [TRH07] improve on this by showing how this can be achieved with a more natural programming style, and including side effects and declassification policies, among which are policies inspired by flow locks. Most recently, Morgernstern and Licata [ML10] show that a rich variety of security policies can be encoded in the dependently typed programming language Agda.

A number of recent expressive languages aimed at expressing a variety of rich security policies do not have information flow control as a primitive notion (as Paragon or Jif). For example, the authorisation policy language Aura can be persuaded to model information flow and declassification polices [JZ09]. Fable [SCH08] focuses on the general idea of label-based policies, allowing user-defined labels and typing constraints (via dependent types). One example is the encoding of a standard information flow lattice policy. A weakness of this approach, according to Swamy et al [SCF$^+$11], is that "verification depends on intricate security proofs too cumbersome for programmers to write down". These concerns are in part addressed by Swamy et al's F* [SCF$^+$11], which is the culmination of a series of languages (from the same group) including Fine [SCC10], FX [BCS11], and F7 [BFG10]. F* is a full-fledged implementation of a dependently typed programming ML-style programming language. An impressive collection of security-specific examples have been encoded in F*, although it may be fair to say that information flow is not naturally modeled in this setting, but has to be encoded using e.g. a monadic approach (c.f. [SCH08]).

The work by Nanevski *et al* [NBG11] can be seen to address this shortcoming (albeit in a purely interactive setting).

## 5.4 Typestate Systems

The concept of *typestate* acknowledges that the runtime state of e.g. an object many times determines which methods are safe to call, which in mainstream object-oriented languages can only be informally specified as documentation for how to use APIs. An example is Java's `File` class, where the method `read()` can only be called if the file has first been opened through a call to `open()`. The purpose of systems with typestate support is to allow formal specification of typestate properties, and enforce that programs correctly follow the specifications.

Paragon implements the concept of typestate properties by its use of lockstate modifiers (section 4.3.1). For instance, a call to `open()` on a file `f` might open a lock `Open(f.id)`, where `id` is an actor field associated with `f`. The `read` method can then specify that it expects `Open(id)` to be open whenever called, which will then be enforced statically by our type system.

One of the main technical challenges of typestate systems is the tracking of aliases to avoid inconsistencies in assumed states when one alias performs a transition. This is exactly the problem of alias tracking for instance actors in Paragon.

The typestate concept was first introduced by Strom and Yemini [SY86], who present an analysis for enforcing typestate properties in existing programming language code. The properties are specified as pre- and post-conditions on statements, where the post-conditions signal the state transitions.

Aldrich et al [ASSS09] introduce *typestate-oriented programming* as an extension of the object-oriented paradigm, and present a Java-like language called *Plaid*. Plaid allows objects to be modeled not in terms of classes, but in terms of the various states they can assume, and allows specification of transitions between states for objects.

Paragon cannot express features that depend on Plaid's first-class states, e.g. "an array of open files", but can otherwise express solutions to their motivating examples.

To facilitate precise alias tracking, Plaid uses a system of *permission modifiers* [BA07] on variables to abstractly describe and restrict how objects may be shared. This reduces the need for many runtime tests, but at the cost of more annotations. Such annotations fit well with the Paragon approach however, and this work could serve as inspiration should instance actor aliasing in Paragon turn out to be a problem in practice.

McGinniss and Gay propose *Hanoi* [MG11] as a practical typestate model for Java programs, requiring no extensions to the language itself. Instead, Hanoi specifications are written separate from the source code, which allows specifications to be written and enforced for pre-existing code.

In terms of expressive power, Paragon's lockstate modifiers can be used to express most Hanoi specifications, with two exceptions. The first is that Hanoi can specify state transitions that are conditional on the returned value of a method. Paragon's runtime querying of locks gives similar expressive power, but only for methods that return boolean values. Second, and related, Hanoi also allows state transitions to be specified for exceptional results of a method. While Paragon can express what state transitions (i.e. lock modifications) that will have taken place *before* an exception is thrown, we cannot state any state transitions that should happen as a result of the throw itself.

Paragon's notion of typestate is more flexible than that in Hanoi in the sense that the different states of an object must not form a tree hierarchy. Parallel aspects of the state of an object can be declared and signaled through the use of different, unrelated lock families, e.g. a bounded queue can be both "not empty", allowing dequeueing, and "not full", allowing enqueueing. Furthermore, Paragon allows state properties to be shared across different classes, since lock families are agnostic to where their actor arguments belong.

McGinniss and Gay list static analysis as an area where Hanoi is lacking. Paragon's type system could be used to statically enforce encodings of Hanoi state specifications, with the exception of transitions based on return values and exceptions as discussed above. However, transitions based on runtime values could not be checked statically regardless. It is also unclear if the current actor alias tracking in Paragon is sufficient.

Typestate-like systems have also been used for security purposes. Walker [Wal00] uses *security automata* to specify policies on what operations are allowed on objects in what states, and transitions between states. His automata can enforce any *safety property*, but not general information flow properties.

# Chapter 6

# Conclusions and Future work

In this thesis we have introduced a full framework for programming with information flow control – the policy specification language Paralocks, an accompanying semantic model, and the programming language Paragon built on top.

We have shown Paralocks to be a flexible and powerful language for specifying dynamic, stateful information flow policies, and have proven it sound with respect to a natural and intuitive semantic security characterisation of information flow. The aim with this work is really two-fold. On the one hand, we have shown the flexibility and usefulness of Paralocks for handling a variety of hands-on information flow challenges. On the other hand, Paralocks is also a very general framework that is capable of expressing and encoding a wide variety of information flow policy mechanisms, and importantly give such mechanisms a concrete information flow semantics. It is our hope and belief that Paralocks can thus serve as a platform that can simplify further research into policy mechanisms, both future and present, and help give a better understanding of the relationship between various mechanisms.

Further, we have shown how Paralocks can be integrated into a full-size object-oriented programming language, by presenting the information flow oriented language Paragon as an extension of Java. We have argued that the combination of Java's encapsulation and the Paralocks policies enables information flow idioms to be nicely encapsulated, so that client applications can freely program with the provided abstractions without the distractions or dangers of seeing their internal representation.

Together, these components cover the whole spectrum from theory to practice, yielding the first information flow framework that is at the same time theoretically sound and useable for solving practical programming problems. Still, much work remains to be done, on many levels.

**Paragon** The first and most pressing task will be to implement the parts of the design that our current prototype does not cover. Things not yet implemented include runtime policies (section 4.3.9), lock properties (section 4.3.2) and null-pointer analysis (section 4.3.6).

A second important task regarding Paragon is to formalise the complete type system, and prove it sound with respect to our Paralocks security condition. Not until that is done can we fully commend ourselves on having produced a complete platform that is proven semantically sound.

Further, we need experience from concrete examples of using information flow libraries, not just defining them! A question one might ask, for example, is whether Paragon is strong enough to provide a drop-in replacement for existing Jif code (modulo minor syntactic issues)?

The language itself - like almost all the related work - is missing a feature which is rather important for modern programming, namely threads. This direction demands both theoretical and practical work.

**Paralocks** Beyond Paragon, there are also reasons to go back and look at ways to improve the underlying Paralocks model. After all, Paragon could be viewed as just one particular implementation of the Paralocks model, and we hope that Paralocks itself will be a valuable contribution to further research in the area of information flow.

One potential direction for future research is to fully exploit the connection to logic-based access control languages. Here we can benefit from various well-behaved extensions to DATALOG such as the addition of stratified negation and constraints on data; see e.g., Becker et al [BFG07] for an elegant authorization language combining such extensions. But the potential here is not just the transferral of technical results. The connection offers new opportunities to transfer *policy concepts* from access control to an information flow context.

On the semantics side, it would be interesting to explore further the "what" dimension of declassification [SS05] and how it can be integrated into our semantic model. Even if this is not the natural focus of the Paralocks language, for the sake of providing a general framework – a *core calculus* of information flow – it would be valuable to semantically characterise such policies, orthogonal to how or whether our Paralocks language could be extended to express them.

Another interesting direction would be to combine Paralocks with a higher-order language like ML or Scala, with a particular focus on type systems to statically enforce Paralocks security in the presence of higher-order functions.

**Concluding remarks** This thesis presents research that improves the state of the art in protecting information security. The work presented herein concerns information flow control, an area that has traditionally not received nearly as much attention as is merited by its importance and usefulness. It is our hope that this work will help bridge the gap between theory and practice concerning information flow control, and help promote awareness of the need for, and the adoption of suitable methods for, programming with information flow.

# Bibliography


[AB05] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, June 2005. Cited on pages 20, 30, 40, 43, 97, 139, and 141.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, 2008. Cited on pages 31, 37, and 38.

[AS07] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007. Cited on pages 22, 31, 35, 44, 45, 46, 47, 61, 139, 141, and 142.

[Asp95] D. Aspinall. Subtyping with singleton types. In *In Eighth International Workshop on Computer Science Logic*, pages 1–15. Springer-Verlag, 1995. Cited on page 104.

[ASSS09] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA Companion*, pages 1015–1022, 2009. Cited on page 144.

[BA07] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, 2007. Cited on page 144.

[BCR08] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, 2008. Cited on pages 26, 30, 43, 44, and 141.

[BCS11] J. Borgstrom, J. Chen, and N. Swamy. Verifying stateful programs with substructural state and Hoare types. In *Proceedings*

*of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, 2011. Cited on page 143.

[BD11] M. Balliu and M. Dam. Epistemic temporal logic for information flow security. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2011. Cited on page 142.

[BEM03] A. Belokosztolszki, DM Eyers, and K. Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003*, pages 99–110, 2003. Cited on page 140.

[BFG07] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–15. IEEE Computer Society, 2007. Cited on pages 140 and 148.

[BFG10] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, 2010. Cited on page 143.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973. Cited on page 17.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353. IEEE Computer Society, 2008. Cited on pages 48 and 142.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, 1998. Cited on page 134.

[Bou05] G. Boudol. On typing information flow. In *ICTAC*, pages 366–380, 2005. Cited on page 139.

[BS06a] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. Technical report, Chalmers University

of Technology and Göteborgs University, May 2006. Extended version of [BS06b]. Cited on pages 20 and 31.

[BS06b] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*. Springer Verlag, 2006. Cited on pages 21, 31, 40, 48, and 152.

[BS09] N. Broberg and D. Sands. Flow-sensitive semantics for dynamic information flow policies. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*, Dublin, June 15 2009. ACM. Cited on page 22.

[BS10] N. Broberg and D. Sands. Paralocks – role-based information flow control and beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010. Cited on pages 22 and 136.

[BS11] N. Broberg and D. Sands. Paragon for practical flow-oriented programming. Draft, July 2011. Cited on page 23.

[BWW08] S. Bandhakavi, W. Winsborough, and M. Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proc. IEEE Computer Security Foundations Symposium*, pages 33–47, 2008. Cited on pages 52, 140, and 141.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog(and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989. Cited on page 84.

[CH04] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004. Cited on page 143.

[CLS+01] M. J. Covington, W. Long, S. Srinivasan, A. K. Dev, M. Ahamad, and G. D. Abowd. Securing context-aware applications using environment roles. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 10–20. ACM, 2001. Cited on page 140.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, pages 77–90, 1977. Cited on page 59.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *ACM Conference on Computer and Communications Security*, pages 198–209, October 2004. Cited on pages 26, 48, and 139.

[CM05] S. Chong and A. C. Myers. Language-based information erasure. In *Proc. IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005. Cited on page 139.

[Coh77] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977. Cited on page 17.

[Coh78] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978. Cited on page 17.

[CV97] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences*, 54(1):61 – 78, 1997. Cited on page 84.

[Dam06] M. Dam. Decidability and proof systems for language-based noninterference relations. In *Proc. ACM Symp. on Principles of Programming Languages*, 2006. Cited on pages 30 and 141.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977. Cited on page 17.

[DEG06] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic noninterference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006. Cited on pages 31 and 141.

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976. Cited on pages 17, 53, and 139.

[Den82] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982. Cited on page 17.

[DeT02] J. DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002. Cited on pages 82 and 141.

[DFK06] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, volume 4130 of *LNCS*. Springer, 2006. Cited on pages 140 and 141.

[EP03] R. Echahed and F. Prost. Handling harmless interference. Technical Report 82, Laboratoire Leibniz, IMAG, June 2003. Cited on pages 30 and 141.

[EP05] R. Echahed and F. Prost. Security policy in a declarative style. In *Proceedings of the* $7^{th}$ *International Conference on Principles and Practice of Declarative Programming (PPDP '05)*, Lisboa, Portugal, July 2005. Cited on pages 30 and 141.

[FSG+01] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001. Cited on page 53.

[GI97] L. Giuri and P. Iglio. Role templates for content-based access control. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 153–159. ACM, 1997. Cited on page 140.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996. Cited on page 110.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982. Cited on pages 17 and 30.

[Ham10] C. Hammer. Experiences with pdg-based ifc. In *Engineering Secure Software and Systems, Second International Symposium*, pages 44–60, 2010. Cited on page 143.

[HS09] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009. Cited on page 143.

[HTHZ05] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Workshop on Foundations of Computer Security*, pages 7–18, June 2005. Cited on page 139.

[Jim01] T. Jim. SD3: A trust management system with certified evaluation. In *Proc. IEEE Symp. on Security and Privacy*, 2001. Cited on pages 82, 140, and 141.

[JZ09] L. Jia and S. Zdancewic. Encoding information flow in aura. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009. Cited on page 143.

[KHHJ08] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *ICISS*, pages 56–70, 2008. Cited on page 139.

[LABW91] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 165–182. ACM, 1991. Cited on page 60.

[LGV+09] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009. Cited on page 142.

[LM08] A. Lux and H. Mantel. Who can declassify? In *Preproceedings of the Workshop on Formal Aspects in Security and Trust (FAST)*, 2008. Cited on pages 30 and 141.

[LMW02] N. Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002. Cited on pages 82, 140, and 141.

[LZ10] P. Li and S. Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci*, 411(19), 2010. Cited on page 143.

[MBL97] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) , Paris, France*, pages 132–145, January 1997. Cited on page 134.

[McC87] D. McCullough. Specifications for multi-level security and hook-up property. In *Proc. IEEE Symp. on Security and Privacy*, pages 161–166, April 1987. Cited on page 39.

[MG11] I. McGinniss and S. J. Gay. Hanoi: A typestate dsl for java. Technical Report TR-2011-326, University of Glasgow, 2011. Cited on page 145.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997. Cited on pages 19, 50, 52, 76, 77, 83, and 139.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998. Cited on pages 50 and 83.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000. Cited on page 50.

[ML10] J. Morgenstern and D. R. Licata. Security-typed programming within dependently-typed programming. In *IProceedings of the 15th ACM SIGPLAN international conference on Functional Programming*, 2010. Cited on page 143.

[MR07] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, March 2007. Cited on pages 30 and 141.

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 129–145. Springer-Verlag, November 2004. Cited on pages 22, 30, 48, and 141.

[MSZ04] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 172–186, June 2004. Cited on pages 26, 49, 91, 132, 133, and 141.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming*

*Languages*, pages 228–241, January 1999. Cited on pages 79 and 132.

[MZZ+06] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2006. Cited on pages 19, 79, 132, and 142.

[NBG11] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proc. IEEE Symp. on Security and Privacy*, 2011. Cited on pages 140 and 143.

[NS10] K. Nakata and A. Sabelfeld. Securing class initialization. In *Trust Management IV - 4th IFIP WG 11.11 International Conference, IFIPTM 2010, Morioka, Japan, June 16-18, 2010. Proceedings*, volume 321. Springer, 2010. Cited on page 134.

[Pin95] S. Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symp. on Security and Privacy*, pages 102–113, May 1995. Cited on page 48.

[PS03] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, January 2003. Cited on pages 18 and 142.

[Rus92] J. M. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, 1992. Cited on page 48.

[SCC10] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *In Proceedings of the European Symposium on Programming (ESOP)*, 2010. Cited on page 143.

[SCF+11] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bharagavan, and J. Yang. Secure distributed programming with value-dependent types. In *The 16th ACM SIGPLAN International Conference on Functional Programming*, 2011. Cited on page 143.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb 1996. Cited on page 52.

[SCH08] N. Swamy, B.J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 369–383, 2008. Cited on page 143.

[SHTZ06] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. IEEE Computer Security Foundations Workshop*, 2006. Cited on pages 52, 140, and 141.

[Sim03] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003. Cited on pages 18, 139, and 142.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004. Cited on page 142.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005. Cited on pages 19, 26, 30, 38, 39, 142, and 148.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27, 1980. Cited on page 59.

[SY86] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. Cited on page 144.

[TRH07] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium*, July 2007. Cited on page 143.

[TZ04] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 179–193, 2004. Cited on pages 50, 133, 139, and 141.

[TZ05] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming*, volume 3444 of *LNCS*, pages 279–294. Springer-Verlag, April 2005. Cited on page 50.

[Ull90] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990. Cited on page 59.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996. Cited on page 17.

[Wal00] D. Walker. A type system for expressive security policies. In *POPL*, pages 254–267, 2000. Cited on page 145.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001. Cited on pages 26, 49, 50, and 132.

[ZM07a] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6, 2007. Cited on pages 69, 90, 109, 135, and 139.

[ZM07b] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007. Cited on page 133.

# Appendix A

# Flow locks: Proofs and auxiliary definitions

## A.1 Type system proofs

**Proof of Proposition 1** We need to show that

$$\mathbf{FLS}(\Sigma, c) \wedge \Sigma \subseteq \Sigma' \Longrightarrow \mathbf{FLS}(\Sigma', c)$$

Assume $\mathbf{FLS}(\Sigma, c)$. That means that for all attackers $A = (\alpha, \Delta)$, and all $A$-low memories $L$, we have that if $(\vec{u}u, \Omega) \in \mathit{Run}_A(\Sigma, c, L)$ then $\Omega \subseteq \Delta \Longrightarrow k_A(\vec{u}u, c, L) = k_A(\vec{u}, c, L)$

We make the following observations for using $\Sigma' \supseteq \Sigma$:

Changing the lock state will not affect control flow of a program, which means there will be a direct one-to-one mapping between elements in the two traces, with the same last element of the output sequence.

For each element $(\vec{u}u, \Omega') \in \mathit{Run}_A(\Sigma', c, L)$ with a corresponding $(\vec{u}u, \Omega) \in \mathit{Run}_A(\Sigma, c, L)$, we will have that $\Omega' \supseteq \Omega$. The larger lock state is because adding more locks at the start can never lead to fewer locks open at any subsequent point in the program.

We can then see that using $\Omega' \supseteq \Omega$ in the implication is less restrictive since, $\Omega' \subseteq \Delta$ will hold for fewer attackers.

**Proof of Proposition 5** We need to show that

$$\Sigma \vdash c \rightsquigarrow w, \Delta \wedge \langle \Sigma, c, M \rangle \xrightarrow{\ell} \langle \Sigma', c', M' \rangle \\ \Longrightarrow \Sigma' \vdash c' \rightsquigarrow w', \Delta' \wedge w' \sqsupseteq w \wedge \Delta' \supseteq \Delta$$

which we do by induction of the height of the typing derivation.

Case $c = x := e$: We know

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and $\langle \Sigma, x := e, M\rangle \xrightarrow{\ell} \langle \Sigma, \mathbf{skip}, M'\rangle$ and can show that $\Sigma \vdash \mathbf{skip} \rightsquigarrow \top, \Sigma$ where $\top \sqsupseteq pol(x)$.

Case $c = \mathbf{open}\ \sigma$: We must have

$$\Sigma \vdash \mathbf{open}\ \sigma \rightsquigarrow \top, \Sigma \cup \{\sigma\}$$

and

$$\langle \Sigma, \mathbf{open}\ \sigma, M\rangle \xrightarrow{\ell} \langle \Sigma \cup \{\sigma\}, \mathbf{skip}, M\rangle$$

and the conclusion follows trivially.

Case $c = \mathbf{close}\ \sigma$: Like the case for $\mathbf{open}\ \sigma$.

Case $c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$: We must have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and

$$\langle \Sigma, \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, M\rangle \xrightarrow{\ell} \langle \Sigma, c_i, M\rangle$$

for some $i \in \{1, 2\}$, and we have that $\Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i$ and $w_i \sqsupseteq w_1 \sqcap w_2$ and $\Sigma_i \supseteq \Sigma_1 \cap \Sigma_2$.

Case $c = \mathbf{while}\ (e)\ c$: We have that

$$\frac{\vdash e : r \quad \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma'}$$

and

$$\langle \Sigma, \mathbf{while}\ \ (e)\ \ c, M\rangle \ \xrightarrow{\ell}\ \langle \Sigma, \mathbf{if}\ \ e\ \ \mathbf{then}\ \ c; \mathbf{while}\ \ (e)\ \ c\ \ \mathbf{else}\ \ \mathbf{skip}, M\rangle$$

We can then construct the following derivation:

$$\frac{\begin{array}{cc} \vdash e : r & \Sigma \vdash \mathbf{skip} \rightsquigarrow \top, \Sigma \\ \Sigma \vdash c; \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' & r \sqsubseteq w \end{array}}{\Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c; \mathbf{while}\ (e)\ c\ \mathbf{else}\ \mathbf{skip} \rightsquigarrow w, \Sigma' \cap \Sigma}$$

To prove that the sequential composition can indeed be typed we continue with

$$\frac{\Sigma \vdash c \rightsquigarrow w, \Sigma'' \quad \Sigma'' \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma'}{\Sigma \vdash c; \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma'}$$

The first premise in this deriviation holds because of subtyping for lock sets, together with the observation that $\Sigma \supseteq \Sigma \cap \Sigma'$. By the subtyping rule we then also know that $\Sigma'' \supseteq \Sigma'$. To show the second premise we observe that

$$\frac{\vdash e : r \quad \Sigma'' \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma'' \vdash \textbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma''}$$

and note that this is an equivalent statement since $\Sigma'' \cap \Sigma' = \Sigma'$ due to $\Sigma'' \supseteq \Sigma'$.

Remains to show that $\Sigma' \vdash c \rightsquigarrow w, \Sigma'$. Since $\Sigma' \supseteq \Sigma' \cap \Sigma$ we can show by subtyping of the original premise that $\Sigma' \vdash c \rightsquigarrow w, \Sigma'''$ where $\Sigma''' \supseteq \Sigma'$. To see that $\Sigma''' = \Sigma'$ we note that by the subtyping rule we have that

$$\Sigma'''\backslash\Sigma' \subseteq \Sigma'\backslash(\Sigma' \cap \Sigma) = \Sigma'\backslash\Sigma$$

and the only way to satisfy that inequation is if $\Sigma'''\backslash\Sigma' = \emptyset$, hence $\Sigma''' \subseteq \Sigma'$ and we are done.

Case $c = c_1; c_2$: We have two cases, either $c_1 = \textbf{skip}$ or $c_1 \neq \textbf{skip}$. In the former case the conclusion follows trivially from the typing derivation and semantic rule, so the interesting case is the latter. We then have that

$$\frac{\Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

and

$$\langle \Sigma, c_1; c_2, M \rangle \xrightarrow{\ell} \langle \Sigma', c_1'; c_2, M' \rangle$$

where the induction hypothesis gives us that

$$\Sigma' \vdash c_1' \rightsquigarrow w_1', \Sigma_1' \wedge w_1' \sqsupseteq w_1 \wedge \Sigma_1' \supseteq \Sigma_1$$

We then have by subtyping that $\Sigma_1' \vdash c_2 \rightsquigarrow w_2, \Sigma_2'$ where $\Sigma_2' \supseteq \Sigma_2$, and thus we have that

$$\Sigma' \vdash c_1'; c_2 \rightsquigarrow w_1' \sqcap w_2, \Sigma_2' \wedge w_1' \sqcap_2 \sqsupseteq_1 \sqcap_2 \wedge \Sigma_2' \supseteq \Sigma_2$$

and we are done.

**Definition 28** (Bounded iteration)**.** We define a bound on iteration of while-loops as follows:

$$[\textbf{while}\ (e)\ c]_0 = \textbf{skip}$$

$$[\textbf{while}\ (e)\ c]_k = \textbf{if}\ e\ \textbf{then}\ c; [\textbf{while}\ (e)\ c]_{k-1}\ \textbf{else skip}$$

**Lemma 3** (Consistent run)**.**
*If* $(\vec{u}, \Delta) \in Run_A(\Sigma, c, M\backslash_A)$ *and*

$$\langle \Sigma, c, M \rangle \Longrightarrow_A \langle \Sigma', c', M' \rangle$$

*then* $(\vec{u}, \Delta) \in Run_A(\Sigma', c', M'\backslash_A)$
*Also, if* $(u\vec{u}u', \Delta) \in Run_A(\Sigma, c, M\backslash_A)$ *and*

$$\langle \Sigma, c, M \rangle \xrightarrow{u}_A \langle \Sigma', c', M' \rangle$$

*then* $(\vec{u}u', \Delta) \in Run_A(\Sigma', c', M'\backslash_A)$

The proof follows directly from the construction of $Run_A(\Sigma, c, M\backslash_A)$. Note also that this extends naturally to the case where we take more than one step and/or produce more than one output along the way.

**Lemma 4** (Context typing)**.** *If* $\Sigma \vdash \mathbb{E}[c] \rightsquigarrow w, \Sigma'$*, then* $\Sigma \vdash c \rightsquigarrow w_c, \Sigma''$ *with* $w \sqsubseteq w_c$*.*

Proof: Straighforward induction on the typing derivation for $\mathbb{E}[c]$.

**Lemma 5** (Deterministic expression evaluation)**.** *If* $\vdash e : r$ *and* $r$ *is visible to* $A$ *and* $\langle e, M \rangle \Downarrow v$ *then* $\forall M' \sim_A M$ *we have that* $\langle e, M' \rangle \Downarrow v$*.*

Proof: By induction on the structure of e.
Case $e = n$: We have $\langle n, M \rangle \Downarrow n$ for all $M$ so the conclusion always holds.
Case $e = x$: We have that $\langle x, M \rangle \Downarrow (M[x])$, and since $pol(x)$ is visible to A and $M' \sim_A M$ we know that $M'[x] = M[x]$.
Case $e = e_1 \oplus e_2$: By the assumption and the type rule for operators we know that $r_1 \sqcup r_2$ is visible to A, which implies that $r_i$ is visible to A, $i \in \{1, 2\}$. We apply the induction hypothesis to the subterms, and combine that with $\oplus$ being deterministic, and we are done.

**Lemma 6** (Silent evaluation)**.** *If* $\Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $w$ *is not visible to* $A$*, then running* $c$ *with any starting memory will not produce any* $A$*-visible output, and will not change the memory in any way visible to* $A$*. Formally,* $\forall M$ *we have either*

$$\langle \Sigma, c, M \rangle \Longrightarrow_A \langle \Sigma', \mathbf{skip}, M' \rangle$$

*with* $M' \sim_A M$*, or* $\langle \Sigma, c, M \rangle \Uparrow_A$

Proof: By induction on the height of the typing derivation of $\Sigma \vdash c$.

Case $c = x := e$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

Since $pol(x)$ is not visible to A, for the transition

$$\langle \Sigma, x := e, M \rangle \xrightarrow{\ell} \langle \Sigma, skip, M[x \mapsto v] \rangle$$

where $\langle e, M \rangle \Downarrow v$, $l$ is not visible to A, and $M[x \mapsto v] \sim_A M$.

Case $c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

Neither $w_1$ nor $w_2$ are visible to A, so we can take a transition step

$$\langle \Sigma, \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, M \rangle \xrightarrow{\tau} \langle \Sigma, c_i, M \rangle$$

using either transition rule for conditionals. We apply the induction hypothesis to the resulting term and we are done.

Case $c = \textbf{while } (e)\ c'$: We have

$$\frac{\vdash e : r \quad \Sigma \cap \Sigma' \vdash c' \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Sigma \vdash \textbf{while } (e)\ c' \rightsquigarrow w, \Sigma' \cap \Sigma}$$

To prove this case we need a contradiction. Assume that running $c$ will produce a first output visible to A on the $k$th iteration, i.e. after first performing $k - 1$ silent iterations. This means that up to the point of the first output, running $c$ will be equivalent to running a bounded iteration $[\textbf{while } (e)\ c']_k$ such that:

$$\langle \Sigma, [\textbf{while } (e)\ c']_k, M \rangle \Longrightarrow_A \langle \Delta, c'; \textbf{skip}, M' \rangle$$

with $M' \sim_A M$. We must then have

$$\langle \Delta, c'; \textbf{skip}, M' \rangle \overset{w}{\Longrightarrow}_A \langle \Delta', c''; \textbf{skip}, M'' \rangle$$

since the output cannot have come from the **skip**. But by the induction hypothesis and the typing of $c$ we know that running $c$ cannot produce any output visible to A, and we have our contradiction.

Case $c = c_1; c_2$: We apply the induction hypothesis to both subterms and we are done.

The remaining cases for $c$ can never produce any output or change the memory so they are trivial.

**Lemma 7** (Deterministic output)**.** *If* $\Sigma \vdash$*,* $M \sim_A N$*,* $\langle \Sigma, c, M\rangle \overset{\vec{u}u}{\Longrightarrow}_A \langle \Sigma', c', M'\rangle$ *and* $\langle \Sigma, c, N\rangle \overset{\vec{u}u}{\Longrightarrow}_A \langle \Sigma'', c'', N'\rangle$ *then* $c' = c''$ *and* $M' \sim_A N'$*.*

Proof: By induction on the length of the transition sequence producing $\vec{u}u$ when running with memory $M$.

Case $c = \mathbb{E}[x := e]$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and we identify two cases:

i) $pol(x)$ is visible to A. We must then have

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\mathbf{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\mathbf{skip}], N[x \mapsto v]\rangle$$

where we have $M[x \mapsto v] \sim_A N[x \mapsto v]$. If this was the final output then we are done, and that forms our base case for the induction. Otherwise we apply the induction hypothesis to the resulting configurations.

ii) $pol(x)$ is not visible to A. We then get

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{\ell} \langle \Sigma, \mathbb{E}[\mathbf{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{\ell'} \langle \Sigma, \mathbb{E}[\mathbf{skip}], N[x \mapsto v']\rangle$$

where neither $l$ nor $l'$ are visible to A, and $M[x \mapsto v] \sim_A N[x \mapsto v']$. We continue by applying the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $r$ is visible to A. Then by the deterministic expression evaluation lemma we have $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$. We must have

$$\langle \Sigma, \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $r$ is not visible to A, which means $w_1 \sqcap w_2$ is not visible to A. Then by the silent evaluation lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], M\rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\textbf{skip}], M'\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\textbf{skip}], N'\rangle$$

where $M' \sim_A M \sim_A N \sim_A N'$. Since the lockstate cannot interfere with the evaluation or output, we can continue by applying the induction hypothesis to the configurations $\langle \Sigma', \mathbb{E}[\textbf{skip}], M'\rangle$ and $\langle \Sigma', \mathbb{E}[\textbf{skip}], N'\rangle$.

All other cases are trivial since only one transition rule applies, and that transition does not change the memory or produce any output. We simply perform that transition and apply the induction hypothesis to the resulting configurations.

**Lemma 8** (Deterministic silent termination)**.** *If* $\Sigma \vdash c$ *and* $M \sim_A N$ *and* $\langle \Sigma, c, M\rangle \Longrightarrow_A \langle \Sigma', \textbf{skip}, M'\rangle$ *then either* $\langle \Sigma, c, N\rangle \Longrightarrow_A \langle \Sigma'', \textbf{skip}, N'\rangle$ *or* $\langle \Sigma, c, N\rangle \Uparrow_A$.

Proof: By induction on the length of the transition sequence leading to termination when running with memory $M$.

Case $c = \textbf{skip}$: This case is only interesting since it forms the base case for the induction. **skip** trivially converges to **skip** in 0 steps with no output.

Case $c = \mathbb{E}[x := e]$: Since we know the computation is silent we must have that $pol(x)$ is not visible to A. We then have

$$\langle \Sigma, \mathbb{E}[x := e], M\rangle \xrightarrow{\ell} \langle \Sigma, \mathbb{E}[\textbf{skip}], M[x \mapsto v]\rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N\rangle \xrightarrow{\ell'} \langle \Sigma, \mathbb{E}[\textbf{skip}], N[x \mapsto v']\rangle$$

for some $v, v'$. We have that neither $l$ nor $l'$ are visible to A, and $M[x \mapsto v] \sim_A N[x \mapsto v']$, and we can apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $r$ is visible to A. Then by the deterministic expression evaluation lemma we have $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$. We must have

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $r$ is not visible to A, which means $w_1 \sqcap w_2$ is not visible to A. Then by the silent evaluation lemma we must have that

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], M\rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\textbf{skip}], M'\rangle$$

and either

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\textbf{skip}], N'\rangle$$

where $M' \sim_A M \sim_A N \sim_A N'$, or $\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \Uparrow_A$. In the latter case we are done, in the former we apply the induction hypothesis to the resulting configurations.

All other cases are trivial since only one transition rule could apply, and that transition does not change the memory nor produce any output. We simply perform that transition and apply the induction hypothesis to the resulting configurations.

**Proof of Theorem 1** , repeated here for convenience. What we want to prove is $\Sigma \vdash c \Longrightarrow \mathbf{FLS_{TI}}(c)$, which expanded means

$$\forall A = (\alpha, \Delta), L, (\vec{u}u, \Omega), (\vec{u}u', \Omega') \in Run_A(\Sigma, c, L)$$

we have that

$$\Delta \supseteq \Omega \Longrightarrow k_A(c, L, \vec{u}u) = k_A(c, L, \vec{u}u')$$

We prove this by showing that we must have $w = w'$, by induction on the length of the computation leading to $\vec{u}u$. We identify two cases:

i) $\vec{u}$ has length greater than 0. Then by the deterministic output lemma, and the fact that we know both computations will produce more output and so cannot diverge, we must have that for $M \sim_A N$:

$$\langle \Sigma, c, M\rangle \xRightarrow{\vec{u}}_A \langle \Sigma', c', M'\rangle$$

and

$$\langle \Sigma, c, N\rangle \xRightarrow{\vec{u}}_A \langle \Sigma'', c', N'\rangle$$

where $M' \sim_A N'$. By the consistent run lemma, subject reduction and non-interference of lockstates we then know that $\Sigma' \vdash c'$ and $(w, \Omega), (w', \Omega'') \in Run_A(\Sigma', c', L')$, where $L'$ is the common $A$-low projection of $M'$ and $N'$, and we can apply the induction hypothesis to get $w = w'$.

ii) $\vec{u}$ has length 0. We then proceed to case on $c$.

Case $c = \mathbb{E}[x := e]$: We have

$$\frac{\vdash e : r \quad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

and we identify two cases:

i) $pol(x)$ is not visible to A. Then

$$\langle \Sigma, \mathbb{E}[x := e], M \rangle \xrightarrow{\ell} \langle \Sigma, \mathbb{E}[\mathbf{skip}], M[x \mapsto v] \rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N \rangle \xrightarrow{\ell'} \langle \Sigma, \mathbb{E}[\mathbf{skip}], N[x \mapsto v'] \rangle$$

We have that neither $l$ nor $l'$ are visible to A, and
$M[x \mapsto v] \sim_A N[x \mapsto v']$, and by the consistent run lemma we must have $(w, \Omega), (w', \Omega') \in Run_A(\Sigma, \mathbb{E}[\mathbf{skip}], L)$ where $L$ is the common $A$-low projection of the resulting memories. We can apply the induction hypothesis to get $w = w'$.

ii) $pol(x)$ is not visible to A. Then the next transition will generate the visible output, so we must have $\Omega = \Omega' = \Sigma$. Then by $r(\Sigma) \sqsubseteq pol(x)$ and $\Delta \supseteq \Sigma$ we know that $r$ is visible to A. Then by the deterministic expression evaluation lemma we know $\langle e, M \rangle \Downarrow v \Longrightarrow \langle e, N \rangle \Downarrow v$, so we must have

$$\langle \Sigma, \mathbb{E}[x := e], M \rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\mathbf{skip}], M[x \mapsto v] \rangle$$

and

$$\langle \Sigma, \mathbb{E}[x := e], N \rangle \xrightarrow{x(v)}_A \langle \Sigma, \mathbb{E}[\mathbf{skip}], N[x \mapsto v] \rangle$$

We have $w = w' = x(v)$ and we are done.

Case $c = \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]$: We have

$$\frac{\vdash e : r \quad \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $r$ is not visible to A. Then by the silent evaluation lemma and $r \sqsubseteq w_1 \sqcap w_2$ we know the subterms cannot produce A-visible output. We must have

$$\langle \Sigma, \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], M \rangle \Longrightarrow_A \langle \Sigma', \mathbb{E}[\mathbf{skip}], M' \rangle$$

and

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \Longrightarrow_A \langle \Sigma'', \mathbb{E}[\textbf{skip}], N'\rangle$$

with $M' \sim_A M \sim_A N \sim_A N'$. By the consistent run lemma we must also have $(w, \Omega), (w', \Omega'') \in \mathit{Run}_A(\Sigma', \mathbb{E}[\textbf{skip}], L)$ and we can apply the induction hypothesis to get $w = w'$.

ii) $r$ is visible to A. Then by the deterministic expression evaluation lemma we know $\langle e, M\rangle \Downarrow v \Longrightarrow \langle e, N\rangle \Downarrow v$ and we must have

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], M\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], M\rangle$$

and

$$\langle \Sigma, \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], N\rangle \xrightarrow{\tau} \langle \Sigma, \mathbb{E}[c_i], N\rangle$$

for some $i \in \{1, 2\}$. By the consistent run lemma we must have $(w, \Omega), (w', \Omega'') \in \mathit{Run}_A(\Sigma, \mathbb{E}[c_i], L)$ and we can apply the induction hypothesis to get $w = w'$.

All other cases are trivial since only one transition rule applies, and that transition does not change the memory or produce any output. We simply perform that transition, note that the consistent run lemma applies, and apply the induction hypothesis to the resulting configurations.

## A.2 DLM encoding

Here we give the full formal details of our encoding of the decentralised label model (DLM). We begin by restating the actual encoding:

To handle the general case of the encoding we need to deal with the case of a *potential reader* (a reader who is a reader for one but not all owners). For these readers we need to consider the owners who do *not* permit $r$ to read the data.

**Definition 29** (Label Encoding)**.** Suppose that $r$ is a (potential or effective) reader for some label $L$, and $O$ is a subset of owners for $L$. We say that the pair $(O, r)$ is a *conflict pair* for label $L$ if

$$O = \{o \mid o \in \mathit{owners}(L), r \notin \mathit{readers}(L, o)\}\,.$$

Intuitively, $O$ are the owners who have not permitted $r$ to read data labelled $L$.

Now we can define the general encoding of Labels as policies $[\![\cdot]\!] : \mathit{Label} \to \mathit{Policy}$ by

$$\begin{aligned}[\![L]\!] = \{&\forall x.\ \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L)\} \Rightarrow x\} \\ &\cup\ \{\forall y.\ \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \mid \\ &\qquad\qquad (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L\}\end{aligned}$$

What we want to show is the following theorem:

**Theorem 5.** $L_1 \sqsubseteq_{DLM} L_2$ if and only if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$.
Further, $[\![L_1 \sqcup_{DLM} L_2]\!] = [\![L_1]\!] \sqcup [\![L_2]\!]$.

We begin with the proof of equivalence for the partial orderings.

We want to prove $L_1 \sqsubseteq_{DLM} L_2$ if and only if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$. We start with the only if direction. We know from $L_1 \sqsubseteq_{DLM} L_2$ that $owners(L_1) \subseteq owners(L_2)$ and that $\forall o.\, readers(L_1, o) \supseteq readers(L_2, o)$. We have that

$$\begin{aligned}[\![L_i]\!] = \{&\forall x.\ \{RunsFor(o) \mid o \in owners(L_i)\} \Rightarrow x\} \\ \cup\ \{&\forall y.\ RunsFor(o_1), \ldots, RunsFor(o_n), ActsFor(r, y) \Rightarrow y \mid \\ &(\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_i\}\end{aligned}$$

To prove that $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$ we must show that $\forall c_2 \in [\![L_2]\!].\, \exists c_1 \in [\![L_1]\!].\, c_1 \sqsubseteq c_2$. Looking at $[\![L_2]\!]$ we can immediately see that

$$\begin{aligned}\{&\forall x.\ \{RunsFor(o) \mid o \in owners(L_1)\} \Rightarrow x\} \\ &\sqsubseteq \{\forall x.\ \{RunsFor(o) \mid o \in owners(L_2)\} \Rightarrow x\}\end{aligned}$$

since we know $owners(L_1) \subseteq owners(L_2)$. All remaining clauses pertain to some particular (actual or potential) reader

$$r \in \bigcup_{o \in owners(L_2)} readers(L_2, o)$$

We then have two cases:

i) $r \in \bigcup_{o \in owners(L_1)} readers(L_1, o)$. We then have that

$$\begin{aligned}\{&\forall y.\ RunsFor(o_1), \ldots, RunsFor(o_n), \\ &\quad ActsFor(r, y) \Rightarrow y \mid\ (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_1\} \\ &\quad \subseteq \{\forall y.\ RunsFor(o_1), \ldots, RunsFor(o_n), ActsFor(r, y) \Rightarrow y \\ &\qquad\qquad \mid\ (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_2\}\end{aligned}$$

since $owners(L_1) \subseteq owners(L_2)$ and $r \notin readers(L_1, o) \Longrightarrow r \notin readers(L_2, o)$.

ii) $r \notin \bigcup_{o \in owners(L_1)} readers(L_1, o)$. Then since $owners(L_1) \subseteq owners(L_2)$ and $r \in readers(L_2, o) \Longrightarrow r \in readers(L_1, o)$ we must have a clause in $[\![L_2]\!]$ of the form $\forall x.\, \Sigma \cup \{ActsFor(r, x)\} \Rightarrow x$ where $\Sigma \supseteq \{RunsFor(o) \mid o \in owners(L_1)\}$. Then we have that

$$\forall x.\ \{RunsFor(o) \mid o \in owners(L_i)\} \Rightarrow x \sqsubseteq \forall x.\, \Sigma \cup \{ActsFor(r, x)\} \Rightarrow x$$

and we are done.

Next we prove the if direction. We know $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$, which means $\forall c_2 \in [\![L_2]\!]. \exists c_1 \in [\![L_1]\!]. c_1 \sqsubseteq c_2$. To prove $L_1 \sqsubseteq_{DLM} L_2$ we need to show that $owners(L_1) \subseteq owners(L_2)$ and $\forall o. readers(L_1, o) \supseteq readers(L_2, o)$. The first is easy. We know $[\![L_2]\!]$ includes the clause

$$\forall x. \{RunsFor(o) \mid o \in owners(L_2)\} \Rightarrow x$$

and the only clause from $[\![L_1]\!]$ that could be less restrictive than this is

$$\forall x. \{RunsFor(o) \mid o \in owners(L_1)\} \Rightarrow x$$

which immediately gives us $owners(L_1) \subseteq owners(L_2)$.

The remaining clauses in $[\![L_2]\!]$ are on the form $\forall y. \Sigma \cup \{ActsFor(r, y)\} \Rightarrow y$ for some $r \in \bigcup_{o \in owners(L_2)} readers(L_2, o)$, where we have that
$\Sigma = \{RunsFor(o) \mid o \in O, (O, r)$ is a conflict pair for $L_2\}$.

If $r \notin \bigcup_{o \in owners(L_1)} readers(L_1, o)$ then we have that

$$\forall x. \{RunsFor(o) \mid o \in owners(L_1)\} \Rightarrow x \sqsubseteq \forall y. \Sigma \cup \{ActsFor(r, y)\} \Rightarrow y$$

which teaches us nothing, so we restrict our attention to the other case. We thus assume $r \in \bigcup_{o \in owners(L_1)} readers(L_1, o)$ which means that $\exists o \in owners(L_1). RunsFor(o) \notin \Sigma$, so we know that

$$\forall x. \{RunsFor(o) \mid o \in owners(L_1)\} \Rightarrow x \not\sqsubseteq \forall y. \Sigma \cup \{ActsFor(r, y)\} \Rightarrow y$$

.

Thus we must have that one of the other clauses in $[\![L_1]\!]$ is less restrictive than the clause at hand. That clause must then be on the form

$$\forall y. \Sigma' \cup \{ActsFor(r, y)\} \Rightarrow y$$

with $\Sigma' \subseteq \Sigma$, and thus we have

$$\begin{aligned}\Sigma' &= \{RunsFor(o) \mid o \in O, (O, r) \text{ is a conflict pair for } L_1\} \\ &\subseteq \{RunsFor(o) \mid o \in O, (O, r) \text{ is a conflict pair for } L_2\} = \Sigma\end{aligned}$$

In other words, $\forall o \in owners(L_1). r \notin readers(L_1, o) \Longrightarrow r \notin readers(L_2, o)$, and thus $readers(L_2, o) \subseteq readers(L_1, o)$ as required. For $o \notin owners(L_1)$ we have that $readers(L_1, o)$ is defined to be the set of all possible readers, so for those the conlcusion trivially holds, and we are done.

Next we want to prove that $[\![L_1]\!] \sqcup [\![L_2]\!] = [\![L_1 \sqcup_{DLM} L_2]\!]$. We have that $[\![L_1]\!] \sqcup [\![L_2]\!]$

$$
\begin{aligned}
=&\{\forall x.\, \{\mathit{RunsFor}(o) \mid\ o \in \mathit{owners}(L_1)\} \Rightarrow x\} \\
&\quad \cup\ \{\forall y.\, \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \\
&\qquad \mid\ (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_1\} \\
&\sqcup \{\forall x.\, \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L_2)\} \Rightarrow x\} \\
&\quad \cup\ \{\forall y.\, \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \\
&\qquad \mid\ (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_2\} \\
=&\{\forall x.\, \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L_1) \cup \mathit{owners}(L_2)\} \Rightarrow x\} \\
&\quad \cup\ \{\forall y.\, \mathit{RunsFor}(o_{11}), \ldots, \mathit{RunsFor}(o_{1n}), \\
&\qquad \mathit{RunsFor}(o_{21}), \ldots, \mathit{RunsFor}(o_{2m}), \mathit{ActsFor}(r, y) \Rightarrow y \\
&\qquad\quad \mid\ (\{o_{11}, \ldots, o_{1n}\}, r) \text{ is a conflict pair for } L_1, \\
&\qquad\quad\ \ (\{o_{21}, \ldots, o_{2m}\}, r) \text{ is a conflict pair for } L_2\} \\
=&\{\forall x.\, \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L_1 \sqcup_{DLM} L_2)\} \Rightarrow x\} \\
&\quad \cup\ \{\forall y.\, \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \\
&\qquad \mid\ (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L_1 \sqcup_{DLM} L_2\} \\
=&[\![L_1 \sqcup_{DLM} L_2]\!]
\end{aligned}
$$

# Appendix B

# Paralocks: Proofs and auxiliary definitions

## B.1 Type System Security Proof

In this section we discuss and prove properties of the type system of our example language for Paralocks, where the climax is the proof that a well-typed program is (termination insensitive) Paralocks secure.

The type system as stated in section 3.5 is slightly problematic to prove things for since some properties of the typing are not *localised*, i.e. some properties that we know hold for whole well-typed programs may not be visible locally from the typing of every subprogram. To help with our proofs we thus begin by giving a type system that is locally stronger than the one presented in section 3.5, and prove that the two are equivalent for the purpose of typing whole programs.

The localised type system can thus be found in figure B.1. There are two differences from the one presented previously. First, it includes a type rule for the internal **for** construct, which differs from the **forall** rule only by not requiring all actor variables to be fresh. Indeed we expect them not to be, since we may be in the middle of looping over the lock set and those variables will have been bound to actors on earlier iterations. Second the localisation part, which amounts to adding a constraint on the use of actor variables in the rules for assignment, **open** and **close**. This constraint says that any actor variables that you use must have less restrictive policies than the entity you use them with, to avoid leaks through observing the change to the entity in question and from that learning the identity of a "secret" actor.

As previously stated, for whole self-contained surface programs (i.e. programs with no **for** constructs), the following property holds:

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash_f \textbf{open}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash_f \textbf{close}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \simeq \vec{b}\}}$$

$$\frac{}{\Lambda; \Sigma \vdash_f \textbf{skip} \rightsquigarrow \top, \Sigma}$$

$$\frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq pol(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])}{\Lambda; \Sigma \vdash_f x[\vec{a}] := e \rightsquigarrow pol(x[\vec{a}]), \Sigma}$$

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash_f c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash_f \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \cap \Sigma' \vdash_f c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma \vdash_f \textbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

$$\frac{\Lambda; \Sigma \vdash_f c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma_1 \vdash_f c_2 \rightsquigarrow w_2, \Sigma_2}{\Lambda; \Sigma \vdash_f c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

$$\frac{\begin{array}{c}\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash_f c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash_f c_2 \rightsquigarrow w_2, \Sigma_2 \\ pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)\end{array}}{\Lambda; \Sigma \vdash \textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash_f c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \vec{a} \cap \Lambda = \emptyset}{\Lambda; \Sigma \vdash_f \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash_f c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash_f \textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \boldsymbol{\Sigma}\ \textbf{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash_f c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash_f \textbf{newactor}\ a\ \textbf{in}\ c \rightsquigarrow \bot, \Sigma' \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}}$$

**Figure B.1:** Localised Paralocks Type System

**Theorem 6.**

$$\emptyset;\emptyset \vdash c \rightsquigarrow w, \Sigma' \iff \emptyset;\emptyset \vdash_f c \rightsquigarrow w, \Sigma'$$

The right-to-left implication trivially holds since the localised type system is strictly more restrictive locally, so we give our attention to the other direction. To prove the left-to-right implication we generalise the statement to the following lemma:

**Lemma 9.** *If* $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'$ *and* $\forall a \in \Lambda.\, pol(a) \sqsubseteq w$ *then* $\Lambda; \Sigma \vdash_f c \rightsquigarrow w, \Sigma'$.

and note that at the top level, with $\Lambda = \emptyset$, the added condition for the actor variables in $\Lambda$ trivially holds. Thus if we can prove this lemma we have also proven the main theorem. We proceed by induction on the height of the typing derivation for $\Lambda; \Sigma \vdash c$.

Case $c = x[\vec{a}] := e$: We have $w = pol(x[\vec{a}])$ and thus the added condition for $\vdash_f$ that $\forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])$ holds.

Case $c = \textbf{open}\ \sigma(\vec{a})$: We have $w = pol(\sigma)$ and thus the added condition for $\vdash_f$ that $\forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)$ holds.

Case $c = \textbf{close}\ \sigma(\vec{a})$: Same as for **open**.

Case $c = \textbf{newactor}\ a\ \textbf{in}\ c'$: The newly created actor variable will have policy $\bot$, so for the enclosed subcomputation the second premise for the induction hypothesis is guaranteed to hold, so we simply call the induction hypothesis on $c'$.

Case $c = \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'$: We know that the write effect of $c'$ is more restrictive than $pol(\sigma)$, which happens to be the policy of all the introduced actor variables. Hence we can call the induction hypothesis on $c'$.

The remaining cases are trivial, just call the induction hypothesis on any subcomputations.

That concludes the proof of equivalene for top-level programs for the two type systems. From this point on we will use the full type system, but sometimes omit the subscript on the turnstile for the sake of readability.

Next we prove that subject reduction preserves typeability of programs. To do that we first need the following lemma:

**Lemma 10** (Subsumption)**.** *If* $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $\Sigma' \supseteq \Sigma$ *then* $\Lambda; \Sigma' \vdash c \rightsquigarrow w, \Delta'$ *where* $\Delta' \supseteq \Delta$ *and* $\Delta' \setminus \Delta \subseteq \Sigma' \setminus \Sigma$.

The proof is straight-forward induction on the height of the typing derivation. We will not go into the details, instead just noting that the only use of $\Sigma$ in any of the type rules is in the rule for assignments, where it appears in a covariant position in the specialisation of the policy of the written data.

Now we can move on to subject reduction proper. Formally we have

**Lemma 11** (Preservation)**.** *Let us say that state $S$ is* compatible with $\Sigma$ *if* $\mathbf{LS}(S) \supseteq \mathbf{Act}(S)(\Sigma)$*. Similarly we say that state $S$ is compatible with an actor set $\Lambda$ if* $\mathrm{dom}(\mathbf{Act}(S)) \supseteq \Lambda$*.*

*Now suppose that* $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$*. Then if* $\Lambda$ *and* $\Sigma$ *are compatible with $S$ then* $\Lambda'; \Sigma' \vdash c' \rightsquigarrow w', \Delta'$ *for some* $\Lambda'$ *and* $\Sigma'$ *compatible with* $S'$*,* $w \sqsubseteq w'$ *and* $\Delta \subseteq \Delta'$*.*

Proof by induction on the height of the typing derivation for $c$.

Case $c = x[\vec{a}] := e$: We know

$$\frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq \mathit{pol}(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, \mathit{pol}(a) \sqsubseteq \mathit{pol}(x[\vec{a}])}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow \mathit{pol}(x[\vec{a}]), \Sigma}$$

and $\langle x[\vec{a}] := e, S\rangle \xrightarrow{\ell} \langle \mathbf{skip}, S'\rangle$. Since $\mathbf{LS}(S) = \mathbf{LS}(S')$ and $\mathbf{Act}(S) = \mathbf{Act}(S')$ we pick $\Lambda' = \Lambda$ and $\Sigma' = \Sigma$ and can show that $\Lambda; \Sigma \vdash \mathbf{skip} \rightsquigarrow \top, \Sigma$ where $\mathit{pol}(x) \sqsubseteq \top$.

Case $c = \mathbf{open}\ \sigma(\vec{a})$: We must have

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, \mathit{pol}(a) \sqsubseteq \mathit{pol}(\sigma)}{\Lambda; \Sigma \vdash \mathbf{open}\ \sigma(\vec{a}) \rightsquigarrow \mathit{pol}(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

and

$$\langle \mathbf{open}\ \sigma(\vec{a}), S\rangle \xrightarrow{\ell} \langle \mathbf{skip}, S \cup \{\sigma(\mathbf{Act}(S)(\vec{a}))\}\rangle$$

We pick $\Sigma' = \Sigma \cup \{\sigma(\vec{a})\}$ and the conclusion follows trivially.

Case $c = \mathbf{close}\ \sigma(\vec{a})$: Like the case for $\mathbf{open}\ \sigma(\vec{a})$.

Case $c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$: We must have

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and

$$\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, S\rangle \xrightarrow{\ell} \langle c_i, S\rangle$$

for some $i \in \{1, 2\}$, and we have that $\Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i$ and $w_i \sqsupseteq w_1 \sqcap w_2$ and $\Sigma_i \supseteq \Sigma_1 \cap \Sigma_2$.

Case $c = \mathbf{while}\ (e)\ c$: We have that

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

and

$$\langle \mathbf{while}\ (e)\ c, S\rangle \xrightarrow{\ell} \langle \mathbf{if}\ e\ \mathbf{then}\ c; \mathbf{while}\ (e)\ c\ \mathbf{else}\ \mathbf{skip}, S\rangle$$

We can then construct the following derivation:

$$\frac{\Lambda \vdash e : r \qquad \Lambda; \Sigma \vdash \mathbf{skip} \rightsquigarrow \top, \Sigma \qquad \Lambda; \Sigma \vdash c; \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \qquad r \sqsubseteq w}{\Lambda; \Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c; \mathbf{while}\ (e)\ c\ \mathbf{else\ skip} \rightsquigarrow w, \Sigma' \cap \Sigma}$$

To prove that the sequential composition can indeed be typed we continue with

$$\frac{\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'' \quad \Lambda; \Sigma'' \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash c; \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma'}$$

The first premise in this deriviation holds because of subsumption for lock sets, together with the observation that $\Sigma \supseteq \Sigma \cap \Sigma'$. By the subsumption lemma we then also know that $\Sigma'' \supseteq \Sigma'$. To show the second premise we observe that

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma'' \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma'' \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma''}$$

and note that this is an equivalent statement since $\Sigma'' \cap \Sigma' = \Sigma'$ due to $\Sigma'' \supseteq \Sigma'$.

Remains to show that $\Lambda; \Sigma' \vdash c \rightsquigarrow w, \Sigma'$. Since $\Sigma' \supseteq \Sigma' \cap \Sigma$ we can show by subsumption of the original premise that $\Lambda; \Sigma' \vdash c \rightsquigarrow w, \Sigma'''$ where $\Sigma''' \supseteq \Sigma'$. To see that $\Sigma''' = \Sigma'$ we note that by the subsumption lemma we have that

$$\Sigma'''\backslash\Sigma' \subseteq \Sigma'\backslash(\Sigma' \cap \Sigma) = \Sigma'\backslash\Sigma$$

and the only way to satisfy that inequation is if $\Sigma'''\backslash\Sigma' = \emptyset$, hence $\Sigma''' \subseteq \Sigma'$ and we are done.

Case $c = c_1; c_2$: We have two cases, either $c_1 = \mathbf{skip}$ or $c_1 \neq \mathbf{skip}$. In the former case the conclusion follows trivially from the typing derivation and semantic rule, so the interesting case is the latter. We then have that

$$\frac{\Lambda; \Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Lambda; \Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

and

$$\langle c_1; c_2, S\rangle \xrightarrow{\ell} \langle c_1'; c_2, S'\rangle$$

where the induction hypothesis gives us that

$$\Lambda'; \Sigma' \vdash c_1' \rightsquigarrow w_1', \Sigma_1' \wedge w_1' \sqsupseteq w_1 \wedge \Sigma_1' \supseteq \Sigma_1$$

We then have by subtyping that $\Lambda'; \Sigma_1' \vdash c_2 \rightsquigarrow w_2, \Sigma_2'$ where $\Sigma_2' \supseteq \Sigma_2$, and thus we have that

$$\Lambda'; \Sigma' \vdash c_1'; c_2 \rightsquigarrow w_1' \sqcap w_2, \Sigma_2' \wedge w_1' \sqcap_2 \sqsupseteq_1 \sqcap_2 \wedge \Sigma_2' \supseteq \Sigma_2$$

and we are done.

Case $c = \textbf{when } \sigma(\vec{a}) \textbf{ do } c_1 \textbf{ else } c_2$. We must have

$$\frac{\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \quad pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \textbf{when } \sigma(\vec{a}) \textbf{ do } c_1 \textbf{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and

$$\langle \textbf{when } \sigma(\vec{a}) \textbf{ do } c_1 \textbf{ else } c_2, S\rangle \xrightarrow{\ell} \langle c_i, S\rangle$$

for some $i \in \{1, 2\}$. If $i = 1$ then we must have that $\sigma(\mathbf{Act}(S)(\vec{a})) \in \mathbf{LS}(S)$ and we can pick $\Sigma' = \Sigma \cup \{\sigma(\vec{a})\}$, and if $i = 2$ we can simply pick $\Sigma' = \Sigma$. In both cases we pick $\Lambda' = \Lambda$ and we have that $\Lambda'; \Sigma' \vdash c_i \rightsquigarrow w_i, \Sigma_i$ and $w_i \sqsupseteq w_1 \sqcap w_2$ and $\Sigma_i \supseteq \Sigma_1 \cap \Sigma_2$.

Case $c = \textbf{newactor } a \textbf{ in } c'$. We must have

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash c \rightsquigarrow w, \Delta}{\Lambda; \Sigma \vdash \textbf{newactor } a \textbf{ in } c \rightsquigarrow \bot, \Delta \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}}$$

and

$$\langle \textbf{newactor } a \textbf{ in } c', S\rangle \xrightarrow{\ell} \langle c', S[a \mapsto \mathbf{a}]\rangle$$

and we can pick $\Lambda' = \Lambda \cup \{a\}$ and $\Sigma' = \Sigma$ and we have that $\Lambda'; \Sigma \vdash c' \rightsquigarrow w, \Delta$ with $w \sqsupseteq \top$ and $\Delta \supseteq \Delta \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}$ as required.

Case $c = \textbf{forall } \sigma(\vec{a}) \textbf{ do } c'$. We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \vec{a} \cap \Lambda = \emptyset}{\Lambda; \Sigma \vdash \textbf{forall } \sigma(\vec{a}) \textbf{ do } c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and

$$\langle \textbf{forall } \sigma(\vec{a}) \textbf{ do } c', S\rangle \xrightarrow{\ell} \langle \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c', S\rangle$$

and we can show that

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c' \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c' \rightsquigarrow \forall \vec{a}.\, w \sqcap pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

Case $c = \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

The case when $\boldsymbol{\Sigma} = \emptyset$ is trivial, so we concentrate on the other case. We then have

$$\langle \textbf{for } \sigma(\vec{a}) \textbf{ in } \{\sigma(\vec{\mathbf{a}})\} \cup \boldsymbol{\Sigma} \textbf{ do } c', S\rangle \xrightarrow{\ell} \langle c'; \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c', S[\vec{a} \mapsto \vec{\mathbf{a}}]\rangle$$

We pick $\Lambda' = \Lambda \cup \vec{a}$ and can construct the following derivation:

$$\frac{\Lambda'; \Sigma \vdash c' \rightsquigarrow w, \Sigma'' \quad \Lambda'; \Sigma'' \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c' \rightsquigarrow pol(\sigma), \Sigma' \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}{\Lambda'; \Sigma \vdash c'; \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c' \rightsquigarrow pol(\sigma), \Sigma' \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

The left premise holds from subtyping, with $\Sigma \supseteq \Sigma \cap \Sigma'$ and thus $\Sigma'' \supseteq \Sigma'$. Then we can construct

$$\frac{\Lambda'; \Sigma'' \cap \Sigma \vdash c' \rightsquigarrow w, \Sigma''}{\Lambda'; \Sigma'' \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c' \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma'' \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and note that $\Sigma'' \cap \Sigma' = \Sigma'$ due to $\Sigma'' \supseteq \Sigma'$. The premise then holds since $\Sigma' \supseteq \Sigma' \cap \Sigma$ so we have

$$\Lambda'; \Sigma' \vdash c' \rightsquigarrow w, \Sigma'''$$

with $\Sigma''' \supseteq \Sigma'$ and $\Sigma''' \setminus \Sigma' \subseteq \Sigma' \setminus \Sigma$. This can only hold if $\Sigma''' \setminus \Sigma' = \emptyset$, so $\Sigma''' \subseteq \Sigma'$, which means $\Sigma''' = \Sigma'$ and we are done, and that concludes the proof of Preservation.

For completeness we also include progress, i.e. that well-typed programs cannot get stuck.

**Definition 30** (Progress)**.** If $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ and $c \neq \textbf{skip}$ and $S$ is compatible with $\Lambda$ and $\Sigma$ then $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$.

Proof: a straight-forward case analysis on commands.

Next we turn our attention to the global properties of the effect components of the type system. We begin with the read effect of expressions, and state the following property:

**Lemma 12** (Deterministic Evaluation)**.** *If $\Lambda \vdash e : r$ and $S$ is compatible with $\Lambda$ and $\mathbf{Act}(S)(r)$ is visible to $A$ and $S \sim_A T$ and $\langle e, S\rangle \Downarrow v$, then $\langle e, T\rangle \Downarrow v$.*

Intuitively this lemma states that if an expression is only computed from locations that are visible to an attacker $A$, then evaluating them with any states that are $A$-equivalent will always produce the same result – in other words, the result cannot depend on data not visible to $A$.

Proof by induction on the height of the typing derivation for $e$.

The case for literal integers is trivial, and the case for operators follows directly from the induction hypothesis and the fact that operators are deterministic.

The only interesting case is thus $e = x[\vec{a}]$. We have

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])}{\Lambda \vdash x[\vec{a}] : pol(x[\vec{a}])}$$

and then $\mathbf{Act}(S)(pol(x[\vec{a}])) = \mathbf{Act}(T)(pol(x[\vec{a}]))$, $x[\mathbf{Act}(S)(\vec{a})] = x[\mathbf{Act}(T)(\vec{a})]$ and then also $\mathbf{Mem}(S)[x[\mathbf{Act}(S)(\vec{a})]] = \mathbf{Mem}(T)[x[\mathbf{Act}(T)(\vec{a})]]$ which are the results of evaluating $e$ with $S$ and $T$ respectively.

Next we turn our attention to the write effect of commands. Before we define the relevant property, we first need the concept of *A-silent divergence*.

**Definition 31** ($A$-silent divergence)**.** We say that a configuration $\langle c, S\rangle$ silently diverges for attacker $A$, written $\langle c, S\rangle \Uparrow_A$, if for all $\langle c', S'\rangle$ reachable from $\langle c, S\rangle$ in zero or more steps we have $\langle c', S'\rangle \xrightarrow{\ell}$ for some $l$ not visible to $A$.

We also need the following lemma for actor mappings:

**Lemma 13** (Actor Mapping Consistency)**.** *If $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'$ and $S$ and $T$ are compatible with $\Lambda$ and $S \sim_A T$ and $\mathbf{Act}(S)(w)$ is visible to $A$, then $\mathbf{Act}(T)(w)$ is visible to $A$.*

*As a corollary, due to symmetry of $\sim_A$, the negative statement also holds, that if $\mathbf{Act}(S)(p)$ is not visible to $A$, then $\mathbf{Act}(T)(p)$ is not visible to $A$ either.*

Proof by induction on the height of the typing derivation.

Case $c = x[\vec{a}] := e$: We have $\forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])$, so if $\mathbf{Act}(S)(pol(x[\vec{a}]))$ is visible to A then all $\mathbf{Act}(S)(pol(a))$ are also visible to A, which in turn means that $\mathbf{Act}(S)$ and $\mathbf{Act}(T)$ must agree on the mapping of those variables due to $S \sim_A T$. Since $pol(x[\vec{a}])$ is closed with respect to the actor variables in $\vec{a}$, we must then have $\mathbf{Act}(S)(pol(x[\vec{a}])) = \mathbf{Act}(T)(pol(x[\vec{a}]))$.

Case $c = \mathbf{open}\ \sigma(\vec{a})$: We have $\forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)$. Since $pol(\sigma)$ is closed, we must have $\mathbf{Act}(S)(pol(\sigma)) = \mathbf{Act}(T)(pol(\sigma))$.

The cases for $\mathbf{close}\ \sigma(\vec{a})$, $\mathbf{forall}\ \sigma(\vec{a})\ \mathbf{do}\ c'$ and $\mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c'$ all follow the same reasoning as the case for $\mathbf{open}\ \sigma(\vec{a})$.

Case $c = \mathbf{newactor}\ a\ \mathbf{in}\ c'$: Trivial since the write effect is $\bot$, and so is always visible.

Case $c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$: We know that $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to A, which means that at least one of $\mathbf{Act}(S)(w_1)$ and $\mathbf{Act}(S)(w_2)$ are visible to A. We call the induction hypothesis on the subterms to get that at least one of $\mathbf{Act}(T)(w_1)$ and $\mathbf{Act}(T)(w_2)$ are visible to A, which in turn means $\mathbf{Act}(T)(w_1 \sqcap w_2)$ is visible to A.

Case $c = c_1; c_2$: Same as for **if**.

Case $c = \mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2$: Same as for **if**.

Case $c = \mathbf{while}\ (e)\ c'$: Simply call the induction hypothesis on the enclosed subterm.

To help with the unrolling of while loops, we also use the following auxiliary definition:

**Definition 32** (Bounded iteration)**.** We define a bound on iteration of while-loops as follows:

$$[\textbf{while } (e)\ c]_0 = \textbf{skip}$$

$$[\textbf{while } (e)\ c]_k = \textbf{if } e \textbf{ then } c; [\textbf{while } (e)\ c]_{k-1} \textbf{ else skip}$$

Now we can state and prove the following global property for the write effects of commands:

**Lemma 14** (Silent evaluation)**.** *If* $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'$ *and* $\Lambda$ *and* $\Sigma$ *are compatible with* $S$ *and and* $\mathbf{Act}(S)(w)$ *is not visible to* $A$*, then either*

$$\langle c, S\rangle \Longrightarrow_A \langle \textbf{skip}, S'\rangle$$

*with* $S \sim_A S'$*, or* $\langle c, S\rangle \Uparrow_A$*.*

Proof by induction on the height of the typing derivation for $c$.

Case $c = \textbf{newactor } a \textbf{ in } c'$: This case cannot be, since the write effect of the command must be $\bot$, which is always visible to $A$.

Case $c = x[\vec{a}] := e$: We must have

$$\frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq pol(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow pol(x[\vec{a}]), \Sigma}$$

so if $\mathbf{Act}(S)(pol(x[\vec{a}]))$ is not visible to $A$ then we must have

$$\langle x[\vec{a}] := e, S\rangle \rightarrow_A \langle \textbf{skip}, S'\rangle$$

with $S \sim_A S'$.

Case $c = \textbf{open } \sigma(\vec{a})$: We must have

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \textbf{open } \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

so if $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A then we must have

$$\langle \textbf{open } \sigma(\vec{a}), S\rangle \rightarrow_A \langle \textbf{skip}, S'\rangle$$

with $S \sim_A S'$.

Case $c = \textbf{close } \sigma(\vec{a})$: Analogous to the case for **open**.

Case $c = \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and we do a separate induction on the size of $\mathbf{\Sigma}$. If $\mathbf{\Sigma} = \emptyset$ then

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \emptyset\ \mathbf{do}\ c', S\rangle \rightarrow_A \langle \mathbf{skip}, S\rangle$$

and we are done. If $\mathbf{\Sigma} \neq \emptyset$ then since $\mathbf{Act}(S)(pol(\sigma))$ is not visible to $A$ we must have

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \{\vec{\mathbf{a}}\} \cup \mathbf{\Sigma}\ \mathbf{do}\ c', S\rangle \rightarrow_A \langle c'; \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c', S'\rangle$$

with $S \sim_A S'$ and we can call the outer induction hypothesis on $c'$, and the inner induction hypothesis on the second subterm.

Note that to call the induction hypothesis here, we must show that $\mathbf{Act}(S')(pol(\sigma))$ is not visible to A, which we get from the actor mapping consistency lemma due to $S \sim_A S'$.

Case $c = \mathbf{while}\ (e)\ c'$: We must have

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma \vdash \mathbf{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

and to prove this we need a contradiction. Assume that running c will produce a first output visible to $A$ on the $k$th iteration, i.e. after first performing $k-1$ silent iterations. This means that up to the point of the first output, running $c$ will be equivalent to running a bounded iteration $[\mathbf{while}\ (e)\ c']_k$ such that:

$$\langle [\mathbf{while}\ (e)\ c']_k, S\rangle \Longrightarrow_A \langle c'; \mathbf{skip}, S'\rangle$$

with $S \sim_A S'$. We must then have

$$\langle c'; \mathbf{skip}, S\rangle \overset{w}{\Longrightarrow}_A \langle c''; \mathbf{skip}, S'\rangle$$

since the output cannot have come from the skip. But by the induction hypothesis and the typing of $c'$ we know that running $c'$ cannot produce any output visible to $A$, and we have our contradiction.

Case $c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$: We have that

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

where neither of $\mathbf{Act}(S)(w_1)$ and $\mathbf{Act}(S)(w_2)$ are visible to $A$, so we take a transition step

$$\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, S\rangle \xrightarrow{\tau} \langle c_i, S\rangle$$

into either branch and apply the induction hypothesis to the resulting term.

Case $c = \mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2$: Same as for **if**.

Case $c = \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'$: Same as for **for**, only with a single silent initial transition step.

Case $c = c_1; c_2$: We simply apply the induction hypothesis to the subterms.

The other global property pertains to the outgoing lockstate component of the typing judgement. Formally we have the following lemma:

**Lemma 15** (Safe Lockstate Approximation)**.** *If* $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ *and* $S$ *is compatible with* $\Lambda$ *and* $\Sigma$ *and*

$$\langle c, S\rangle \stackrel{\vec{u}}{\Longrightarrow} \langle \textbf{skip}, S'\rangle$$

*then* $S'$ *is compatible with* $\Delta$*.*

The proof follows immediately from preservation, since we must have that $\exists \Lambda', \Sigma'.\ S'$ is compatible with $\Lambda'$ and $\Sigma'$ and $\Lambda'; \Sigma' \vdash \textbf{skip} \rightsquigarrow \top, \Sigma'$ where $\Sigma' \supseteq \Delta$. Since the larger the lock set, the harder it is for a state to be compatible with it, we must then also have that $S'$ is compatible with $\Delta$, and we are done.

Finally we tackle the most important proof, namely that the type system guarantees (termination insensitive) Paralocks security for programs. As usual we start by defining a set of auxiliary helper lemmas.

The first of these pertains to runs, and how a run is consistent throughout execution of a program:

**Lemma 16** (Consistent run)**.** *If* $(\vec{u}, \Delta) \in \mathit{Run}(\Lambda, \Sigma, c, L)$ *and* $S$ *is compatible with* $\Lambda$ *and* $\Sigma$ *and* $S \sim_A L$ *and*

$$\langle c, S\rangle \Longrightarrow_A \langle c', S'\rangle$$

*then* $\exists \Lambda', \Sigma'.\ S'$ *is compatible with* $\Lambda'$ *and* $\Sigma'$ *and* $(\vec{u}, \Delta)$
$\in \mathit{Run}(\Lambda', \Sigma', c', \mathbf{Mem}(S')\backslash_A)$.

*Also, if* $(u\vec{u}u', \Delta) \in \mathit{Run}_A(\Lambda, \Sigma, c, L)$ *and*

$$\langle c, S\rangle \stackrel{u}{\rightarrow}_A \langle c', S'\rangle$$

*then* $\exists \Lambda', \Sigma'.\ S'$ *is compatible with* $\Lambda'$ *and* $\Sigma'$ *and* $(\vec{u}u', \Delta)$
$\in \mathit{Run}_A(\Lambda', \Sigma', c', \mathbf{Mem}(S')\backslash_A)$.

The proof follows directly from the construction of $\mathit{Run}_A(\Lambda, \Sigma, c, L)$. Note also that this extends naturally to the case where we take more than one step and/or produce more than one output along the way.

Next we introduce, for simplicity, evaluation contexts as a short-hand on the following form:

**Definition 33** (Evaluation Contexts)**.** $\mathbb{E}[\cdot] = \mathbb{E}[\cdot]\ ;\ c \mid [\cdot]$

Further, we need the following lemma for how how to reason about typing of commands that include evaluation contexts:

**Lemma 17** (Context typing)**.** *If* $\Lambda; \Sigma \vdash \mathbb{E}[c] \rightsquigarrow w, \Sigma'$*, then* $\Lambda; \Sigma \vdash c \rightsquigarrow w_c, \Sigma''$ *with* $w \sqsubseteq w_c$*.*

Proof: Straighforward induction on the typing derivation for $\mathbb{E}[c]$.

Next we have two related lemmas regarding deterministic execution of commands:

**Lemma 18** (Deterministic Execution)**.**
*If* $\Lambda; \Sigma \vdash c$ *and* $S \sim_A T$ *and* $S$ *and* $T$ *are compatible with* $\Lambda$ *and* $\Sigma$ *and* $\langle c, S\rangle \overset{\vec{u}u}{\Longrightarrow}_A \langle c', S'\rangle$ *and* $\langle c, T\rangle \overset{\vec{u}u}{\Longrightarrow}_A \langle c'', T'\rangle$ *then* $c' = c''$ *and* $S' \sim_A T'$*.*

Proof: By induction on the length of the transition sequence producing $\vec{u}u$ when running with state $S$. Throughout the proof we implicitly use the preservation lemma to ensure that resulting configurations actually fulfill the premise for the induction hypothesis. We will also implicitly use the actor mapping consistency lemma to split each case into two cases based on attacker visibility.

Case $c = \mathbb{E}[x[\vec{a}] := e]$: We have

$$\frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq pol(x[\vec{a}]) \quad \forall a \in \vec{a}.\, a \in \Lambda}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow pol(x[\vec{a}]), \Sigma}$$

and we identify two cases:

i) $\mathbf{Act}(S)(pol(x[\vec{a}]))$ is visible to A. We must then have

$$\langle \mathbb{E}[x[\vec{a}] := e], S\rangle \xrightarrow{x(v)}_A \langle \mathbb{E}[\mathbf{skip}], S[x[\vec{a}] \mapsto v]\rangle$$

and

$$\langle \mathbb{E}[x[\vec{a}] := e], T\rangle \xrightarrow{x(v)}_A \langle \mathbb{E}[\mathbf{skip}], T[x[\vec{a}] \mapsto v]\rangle$$

where we have $S[x[\vec{a}] \mapsto v] \sim_A T[x[\vec{a}] \mapsto v]$. If this was the final output then we are done, and that forms a base case for the induction. Otherwise we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(x[\vec{a}]))$ is not visible to A. We then get

$$\langle \mathbb{E}[x[\vec{a}] := e], S\rangle \xrightarrow{\ell} \langle \mathbb{E}[\mathbf{skip}], S[x[\vec{a}] \mapsto v]\rangle$$

and

$$\langle \mathbb{E}[x[\vec{a}] := e], T\rangle \xrightarrow{\ell'} \langle \mathbb{E}[\mathbf{skip}], T[x[\vec{a}] \mapsto v']\rangle$$

where neither $l$ nor $l'$ are visible to A, and $S[x[\vec{a}] \mapsto v] \sim_A T[x[\vec{a}] \mapsto v']$. We continue by applying the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]$: We have

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\textbf{Act}(S)(w_1 \sqcap w_2)$ is visible to A. The $\textbf{Act}(S)(r)$ must also be visible to A, and by the deterministic expression evaluation lemma we have $\langle e, S\rangle \Downarrow v \implies \langle e, T\rangle \Downarrow v$. We must have

$$\langle \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $\textbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

where $S' \sim_A S \sim_A T \sim_A T'$ and we continue by applying the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{open } \sigma(\vec{a})]$: We have

$$\frac{\forall a \in \vec{a}.\, a \in \Lambda \wedge pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \textbf{open } \sigma(\vec{a}) \rightsquigarrow pol(()\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

and we identify two cases:

i) $\textbf{Act}(S)(pol(\sigma))$ is visible to A. Then we must have

$$\langle \mathbb{E}[\textbf{open } \sigma(\vec{a})], S\rangle \xrightarrow{\textbf{open } \sigma(\vec{a})}_A \langle \mathbb{E}[\textbf{skip}], S \cup \{\textbf{Act}(S)(\sigma(\vec{a}))\}\rangle$$

and

$$\langle \mathbb{E}[\textbf{open } \sigma(\vec{a})], T\rangle \xrightarrow{\textbf{open } \sigma(\vec{a})}_A \langle \mathbb{E}[\textbf{skip}], T \cup \{\textbf{Act}(T)(\sigma(\vec{a}))\}\rangle$$

with $S \cup \{\mathbf{Act}(S)(\sigma(\vec{a}))\} \sim_A T \cup \{\mathbf{Act}(T)(\sigma(\vec{a}))\}$. If this was the last output then we are done, and this forms another base case for the induction. Otherwise we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. Then we must have

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], S\rangle \rightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], T\rangle \rightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

with $S' \sim_A T'$ and we call the induction hypothesis on the resulting configurations.

Case $c = \mathbb{E}[\mathbf{while}\ (e)\ c']$: There is only one possible transition, which does not produce any output or affect the state in any way, hence we trivially call the induction hypothesis on the resulting configurations.

Case $c = \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2]$: We must have

$$\frac{\begin{array}{c}\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \\ pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)\end{array}}{\Lambda; \Sigma \vdash \mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to A, which means $\mathbf{Act}(S)(pol(\sigma))$ is also visible to A. Then all $\mathbf{Act}(S)(pol(a))$ are also visible to A, and we must have that if $\mathbf{Act}(S)(\sigma(\vec{a})) \in \mathbf{LS}(S)$ then $\mathbf{Act}(T)(\sigma(\vec{a})) \in \mathbf{LS}(T)$. We will then have

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

where $S' \sim_A S \sim_A T \sim_A T'$ and we continue by applying the induction hypothesis to the resulting configurations.

Case $c = \textbf{newactor}\ a\ \textbf{in}\ c'$: We must have

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash \textbf{newactor}\ a\ \textbf{in}\ c \rightsquigarrow \bot, \Sigma' \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}}$$

and we know that A can see $pol(a) = \bot$, so we must have

$$\langle \mathbb{E}[\textbf{newactor}\ a\ \textbf{in}\ c'], S\rangle \xrightarrow{a(\mathbf{a})}_A \langle \mathbb{E}[c'], S[a \mapsto \mathbf{a}]\rangle$$

and

$$\langle \mathbb{E}[\textbf{newactor}\ a\ \textbf{in}\ c'], T\rangle \xrightarrow{a(\mathbf{a})}_A \langle \mathbb{E}[c'], T[a \mapsto \mathbf{a}]\rangle$$

We have $S[a \mapsto \mathbf{a}] \sim_A T[a \mapsto \mathbf{a}]$ as required. If this was the last output then we are done, otherwise we call the induction hypothesis on the resulting configurations.

Case $c = \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \forall a \in \vec{a}.\, a \notin \Lambda}{\Lambda; \Sigma \vdash \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow \forall \vec{a}.\, w \sqcap pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and we have two cases.

i) $\mathbf{Act}(S)(pol(\sigma))$ is visible to A. Then we must have that $\{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(S)\} = \{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(T)\} = \mathbf{\Sigma}$ and we have

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c'], S\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c'], T\rangle$$

and we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. Then by the silent execution lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], S\rangle \Longrightarrow \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], T\rangle \Longrightarrow \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$ and we apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c' \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c' \rightsquigarrow \forall \vec{a}.\, w \sqcap pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

where the case where we have $\mathbf{\Sigma} = \emptyset$ lets us trivially apply the induction hypothesis. We thus concentrate on the other case, and then identify two cases:

i) $\mathbf{Act}(S)(pol(\sigma))$ is visible to A, and that policy is inherited by all the actor variables $a$, so we must have

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma} \cup \vec{\mathbf{a}}\ \mathbf{do}\ c', S\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})}_A \langle c'; \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c', S[\vec{a} \mapsto \vec{\mathbf{a}}]\rangle$$

and

$$\langle \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma} \cup \vec{\mathbf{a}}\ \mathbf{do}\ c', T\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})}_A \langle c'; \mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c', T[\vec{a} \mapsto \vec{\mathbf{a}}]\rangle$$

where we have $S[\vec{a} \mapsto \vec{\mathbf{a}}] \sim_A T[\vec{a} \mapsto \vec{\mathbf{a}}]$. If this was the last output then we are done, else we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. Then by the silent execution lemma, and the fact that we know the computations cannot silently diverge before producing the output we seek, we must have that

$$\langle \mathbb{E}[\mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c'], S\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{for}\ \sigma(\vec{a})\ \mathbf{in}\ \mathbf{\Sigma}\ \mathbf{do}\ c'], T\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$, and we can apply the induction hypothesis to the resulting configurations.

**Lemma 19** (Deterministic silent termination)**.**
*If* $\Lambda; \Sigma \vdash c$ *and* $S \sim_A T$ *and* $S$ *and* $T$ *are compatible with* $\Lambda$ *and* $\Sigma$ *and*

$$\langle c, S\rangle \Longrightarrow_A \langle \mathbf{skip}, S'\rangle$$

*then either*

$$\langle c, T\rangle \Longrightarrow_A \langle \mathbf{skip}, T'\rangle$$

*or* $\langle c, T\rangle \Uparrow_A$.

Proof: By induction on the length of the transition sequence leading to termination when running with state $S$.

Case $c = \mathbf{skip}$: This case is only interesting since it forms the base case for the induction. **skip** trivially converges to **skip** in 0 steps with no output.

Case $c = \mathbb{E}[x[\vec{a}] := e]$: Since we know the computation is silent we must have that $\mathbf{Act}(S)(pol(x))$ is not visible to A. We then have

$$\langle \mathbb{E}[x[\vec{a}] := e], S\rangle \xrightarrow{\ell} \langle \mathbb{E}[\mathbf{skip}], S[x[\vec{a}] \mapsto v]\rangle$$

and

$$\langle \mathbb{E}[x[\vec{a}] := e], T\rangle \xrightarrow{\ell'} \langle \mathbb{E}[\mathbf{skip}], T[x[\vec{a}] \mapsto v']\rangle$$

for some $v, v'$. We have that neither $l$ nor $l'$ are visible to A, and $S[x[\vec{a}] \mapsto v] \sim_A T[x[\vec{a}] \mapsto v']$, and we can apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]$: We have

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to A, which means $r$ is also visible to A. Then by the deterministic expression evaluation lemma we have $\langle e, S\rangle \Downarrow v \Longrightarrow \langle e, T\rangle \Downarrow v$. We must have

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma we must have that

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and either

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

where $S' \sim_A S \sim_A T \sim_A T'$, or $\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], T\rangle \Uparrow_A$. In the latter case we are done, in the former we apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})]$: We have

$$\frac{\forall a \in \vec{a}.\, a \in \Lambda \wedge pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \mathbf{open}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

Since we know the computation is silent we must have that $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. We then have

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], S\rangle \xrightarrow{\ell} \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], T\rangle \xrightarrow{\ell'} \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

where neither $l$ nor $l'$ are visible to A, and $S' \sim_A T'$. We continue by applying the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{close}\ \sigma(\vec{a})]$: Same as for **open**.

Case $c = \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2]$: We must have

$$\frac{\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \quad pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to A, which means $\mathbf{Act}(S)(pol(\sigma))$ is also visible to A. Then all $\mathbf{Act}(S)(pol(a))$ are also visible to A, and we must have that if $\mathbf{Act}(S)(\sigma(\vec{a})) \in \mathbf{LS}(S)$ then $\mathbf{Act}(T)(\sigma(\vec{a})) \in \mathbf{LS}(T)$. We will then have

$$\langle \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for the same $i \in \{1, 2\}$. We continue by applying the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma we must have that

$$\langle \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and either

$$\langle \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

or $\langle \mathbb{E}[\textbf{when}\ \sigma(\vec{a})\ \textbf{do}\ c_1\ \textbf{else}\ c_2], T\rangle \Uparrow_A$. In the latter case we are done, in the former we continue by applying the induction hypothesis to the resulting configurations.

Case $c = \textbf{newactor}\ a\ \textbf{in}\ c'$: This case cannot be since it would begin evaluation with a visible output.

Case $c = \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \forall a \in \vec{a}.\, a \notin \Lambda}{\Lambda; \Sigma \vdash \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow \forall \vec{a}.\, w \sqcap pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and we have two cases.

i) $\mathbf{Act}(S)(pol(\sigma))$ is visible to A. Then we must have that $\{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(S)\} = \{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(T)\} = \boldsymbol{\Sigma}$ and we have

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \boldsymbol{\Sigma}\ \textbf{do}\ c'], S\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall } \sigma(\vec{a}) \textbf{ do } c'], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'], T\rangle$$

and we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. Then by the silent execution lemma we must have that

$$\langle \mathbb{E}[\textbf{forall } \sigma(\vec{a}) \textbf{ do } c'], S\rangle \Longrightarrow \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall } \sigma(\vec{a}) \textbf{ do } c'], T\rangle \Longrightarrow \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

or $\langle \mathbb{E}[\textbf{forall } \sigma(\vec{a}) \textbf{ do } c'], T\rangle \Uparrow_A$. In the latter case we are done, in the former we apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c' \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c' \rightsquigarrow \forall \vec{a}.\, w \sqcap pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

where the case where we have $\boldsymbol{\Sigma} = \emptyset$ lets us trivially apply the induction hypothesis. We thus concentrate on the other case, and note that since the evaluation is silent we must have that $\mathbf{Act}(S)(pol(\sigma))$ is not visible to A. Then by the silent execution lemma we must have that

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'], S\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'], T\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

or $\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \boldsymbol{\Sigma} \textbf{ do } c'], T\rangle \Uparrow_A$. In the latter case we are done, in the former we can apply the induction hypothesis to the resulting configurations.

With all this in hand, we can finally turn our attention to the main theorem. Here we give the fully general version of the theorem and proof:

**Theorem 7.** *If $\Lambda; \Sigma \vdash c$ then $c$ is termination-insensitive $(\Lambda, \Sigma)$ Paralocks secure ($\mathbf{PLS_{TI}}(\Lambda, \Sigma, c)$).*

Expanded this means

$$\forall A, L, (\vec{u}u, \boldsymbol{\Delta}), (\vec{u}u', \boldsymbol{\Delta}') \in Run_A(\Lambda, \Sigma, c, L)$$

we have that

$$\mathbf{Cap}(A) \supseteq \boldsymbol{\Delta} \Longrightarrow k_A(c, L, \vec{u}u) = k_A(c, L, \vec{u}u')$$

We prove this by showing that we must have $w = w'$, by induction on the length of the computation leading to $\vec{u}u$. We identify two cases:

i) $\vec{u}$ has length greater than 0. Then by the deterministic execution lemma, and the fact that we know both computations will produce more output and so cannot diverge, we must have that for $S \sim_A T$:

$$\langle c, S\rangle \stackrel{\vec{u}}{\Longrightarrow}_A \langle c', S'\rangle$$

and

$$\langle c, T\rangle \stackrel{\vec{u}}{\Longrightarrow}_A \langle c', T'\rangle$$

where $S' \sim_A T'$. By the consistent run lemma and subject reduction we then know that $\exists \Lambda', \Sigma'.\ \Lambda'; \Sigma' \vdash c'$ and $(w, \mathbf{\Delta}), (w', \mathbf{\Delta}'') \in \mathit{Run}_A(\Lambda', \Sigma', c', L')$, where $L'$ is the common $A$-low projection of $S'$ and $T'$, and we can apply the induction hypothesis to get $w = w'$.

ii) $\vec{u}$ has length 0. We then proceed to case on $c$.

Case $c = \mathbb{E}[x[\vec{a}] := e]$: We have

$$\frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq \mathit{pol}(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, \mathit{pol}(a) \sqsubseteq \mathit{pol}(x[\vec{a}])}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow \mathit{pol}(x[\vec{a}]), \Sigma}$$

and we identify two cases:

i) $\mathbf{Act}(S)(\mathit{pol}(x[\vec{a}]))$ is not visible to $A$. Then

$$\langle \mathbb{E}[x[\vec{a}] := e], S\rangle \xrightarrow{\ell} \langle \mathbb{E}[\mathbf{skip}], S[x[\vec{a}] \mapsto v]\rangle$$

and

$$\langle \mathbb{E}[x[\vec{a}] := e], T\rangle \xrightarrow{\ell'} \langle \mathbb{E}[\mathbf{skip}], T[x[\vec{a}] \mapsto v']\rangle$$

We have that neither $l$ nor $l'$ are visible to A, and $S[x \mapsto v] \sim_A T[x \mapsto v']$, and by the consistent run lemma we must have $(w, \mathbf{\Delta}), (w', \mathbf{\Delta}') \in$ $\mathit{Run}_A(\Lambda, \Sigma, \mathbb{E}[\mathbf{skip}], L)$ where $L$ is the common $A$-low projection of the resulting states. We can apply the induction hypothesis to get $w = w'$.

ii) $\mathbf{Act}(S)(\mathit{pol}(x[\vec{a}]))$ is visible to $A$. Then the next transition will generate the visible output, i.e.

$$\langle \mathbb{E}[x[\vec{a}] := e], S\rangle \xrightarrow{x(v)}_A \langle \mathbb{E}[\mathbf{skip}], S[x[\vec{a}] \mapsto v]\rangle$$

so we must have $\mathbf{\Delta} = \mathbf{LS}(S)$. By $r(\Sigma) \sqsubseteq \mathit{pol}(x[\vec{a}])$ we know $\mathbf{Act}(S)(r(\Sigma))$ is visible to $A$. But since $\mathbf{Cap}(A) \supseteq \mathbf{\Delta} = \mathbf{LS}(S) \supseteq \mathbf{Act}(S)(\Sigma)$ we know that $\mathbf{Act}(S)(r)$ alone is visible to A, without specialisation. Then by the deterministic expression evaluation lemma we know $\langle e, S\rangle \Downarrow v \Longrightarrow \langle e, T\rangle \Downarrow v$, so we must have

$$\langle \mathbb{E}[x[\vec{a}] := e], T\rangle \xrightarrow{x(v)}_A \langle \mathbb{E}[\mathbf{skip}], T[x[\vec{a}] \mapsto v]\rangle$$

We have $w = w' = x(v)$ and we are done.

Case $c = \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})]$: We have

$$\frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \mathbf{open}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}}$$

and we identify two cases:

i) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to $A$. Then

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], S\rangle \xrightarrow{\ell} \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], T\rangle \xrightarrow{\ell'} \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

We have that neither $l$ nor $l'$ are visible to A, and $S' \sim_A T'$, and by the consistent run lemma we must have $(w, \mathbf{\Delta}), (w', \mathbf{\Delta}') \in Run_A(\Lambda, \Sigma, \mathbb{E}[\mathbf{skip}], L)$ where $L$ is the common $A$-low projection of the resulting states. We can apply the induction hypothesis to get $w = w'$.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is visible to $A$. Then the next transition will generate the visible output, i.e.

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], S\rangle \xrightarrow{\mathbf{open}\ \sigma(\vec{\mathbf{a}})}_A \langle \mathbb{E}[\mathbf{skip}], S \cup \{\sigma(\vec{\mathbf{a}})\}\rangle$$

so we must have $\mathbf{\Delta} = \mathbf{LS}(S)$. Since $\forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)$ we know that all $\mathbf{Act}(S)(pol(a))$ is visible to $A$, and hence $\mathbf{Act}(S)(\vec{a}) = \vec{\mathbf{a}} = \mathbf{Act}(T)(\vec{a})$, and thus

$$\langle \mathbb{E}[\mathbf{open}\ \sigma(\vec{a})], T\rangle \xrightarrow{\mathbf{open}\ \sigma(\vec{\mathbf{a}})}_A \langle \mathbb{E}[\mathbf{skip}], T \cup \{\sigma(\vec{\mathbf{a}})\}\rangle$$

We have $w = w' = \mathbf{open}\ \sigma(\vec{\mathbf{a}})$ and we are done.

Case $c = \mathbb{E}[\mathbf{close}\ \sigma(\vec{a})]$: Same as for **open**.

Case $c = \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]$: We have

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma we know that the subterms cannot produce A-visible output. We must have

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$. By the consistent run lemma we must also have $(w, \boldsymbol{\Delta}), (w', \boldsymbol{\Delta}'') \in \mathit{Run}_A(\Lambda', \Sigma', \mathbb{E}[\mathbf{skip}], L)$ and we can apply the induction hypothesis to get $w = w'$.

ii) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to $A$, which means $\mathbf{Act}(S)(r)$ is also visible to $A$. Then by the deterministic expression evaluation lemma we know $\langle e, S\rangle \Downarrow v \Longrightarrow \langle e, T\rangle \Downarrow v$ and we must have

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for some $i \in \{1, 2\}$. By the consistent run lemma we must have $(w, \boldsymbol{\Delta}), (w', \boldsymbol{\Delta}'')$ $\in$ $\mathit{Run}_A(\Lambda, \Sigma, \mathbb{E}[c_i], L)$ and we can apply the induction hypothesis to get $w = w'$.

Case $c = \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2]$: We have

$$\frac{\begin{array}{c}\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \\ \mathit{pol}(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, \mathit{pol}(a) \sqsubseteq \mathit{pol}(\sigma)\end{array}}{\Lambda; \Sigma \vdash \mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

and we identify two cases:

i) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is not visible to A. Then by the silent evaluation lemma we know that the subterms cannot produce A-visible output. We must have

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], S\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], T\rangle \Longrightarrow_A \langle \mathbb{E}[\mathbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$. By the consistent run lemma we must also have $(w, \boldsymbol{\Delta}), (w', \boldsymbol{\Delta}'') \in \mathit{Run}_A(\Lambda', \Sigma', \mathbb{E}[\mathbf{skip}], L)$ and we can apply the induction hypothesis to get $w = w'$.

ii) $\mathbf{Act}(S)(w_1 \sqcap w_2)$ is visible to $A$, which means $\mathbf{Act}(S)(\mathit{pol}(\sigma))$ is also visible to $A$. Then since $\forall a \in \vec{a}.\, \mathit{pol}(a) \sqsubseteq \mathit{pol}(\sigma)$, if $\mathbf{Act}(S)(\sigma(\vec{a})) \in \mathbf{LS}(S)$ then $\mathbf{Act}(T)(\sigma(\vec{a})) \in \mathbf{LS}(T)$, so we must have

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], S\rangle$$

and

$$\langle \mathbb{E}[\mathbf{when}\ \sigma(\vec{a})\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[c_i], T\rangle$$

for some $i \in \{1, 2\}$. By the consistent run lemma we must have $(w, \boldsymbol{\Delta}), (w', \boldsymbol{\Delta}'')$ $\in$ $\mathit{Run}_A(\Lambda, \Sigma, \mathbb{E}[c_i], L)$ and we can apply the induction hypothesis to get $w = w'$.

Case $c = \textbf{newactor}\ a\ \textbf{in}\ c'$: We must have

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash \textbf{newactor}\ a\ \textbf{in}\ c \rightsquigarrow \bot, \Sigma' \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}}$$

and since $pol(a) = \bot$ we must have that $\mathbf{Act}(S)(\bot)$ is visible to $A$. Then we must have

$$\langle \mathbb{E}[\textbf{newactor}\ a\ \textbf{in}\ c'], S\rangle \xrightarrow{a(\mathbf{a})} A\langle \mathbb{E}[c'], S[a \mapsto \mathbf{a}]\rangle$$

and

$$\langle \mathbb{E}[\textbf{newactor}\ a\ \textbf{in}\ c'], T\rangle \xrightarrow{a(\mathbf{a})} A\langle \mathbb{E}[c'], T[a \mapsto \mathbf{a}]\rangle$$

with $S[a \mapsto \mathbf{a}] \sim_A T[a \mapsto \mathbf{a}]$ and we are done.

Case $c = \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \vec{a} \cap \Lambda = \emptyset}{\Lambda; \Sigma \vdash \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and we identify two cases:

i) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to $A$. Then by the silent execution lemma we must have that

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], S\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], T\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$ and we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is visible to $A$. Then we have that

$$\{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(S)\} = \{\sigma(\vec{\mathbf{a}}) \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(T)\} = \mathbf{\Sigma}$$

and we must have

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], S\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c'], S\rangle$$

and

$$\langle \mathbb{E}[\textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c'], T\rangle \xrightarrow{\tau} \langle \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c'], T\rangle$$

and we apply the induction hypothesis to the resulting configurations.

Case $c = \mathbb{E}[\textbf{for}\ \sigma(\vec{a})\ \textbf{in}\ \mathbf{\Sigma}\ \textbf{do}\ c']$: We must have

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w}{\Lambda; \Sigma \vdash \textbf{forall}\ \sigma(\vec{a})\ \textbf{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \emptyset\}}$$

and we identify two cases:

i) $\mathbf{Act}(S)(pol(\sigma))$ is not visible to $A$. Then by the silent execution lemma we must have that

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \mathbf{\Sigma} \textbf{ do } c'], S\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], S'\rangle$$

and

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \mathbf{\Sigma} \textbf{ do } c'], T\rangle \Longrightarrow_A \langle \mathbb{E}[\textbf{skip}], T'\rangle$$

with $S' \sim_A S \sim_A T \sim_A T'$ and we apply the induction hypothesis to the resulting configurations.

ii) $\mathbf{Act}(S)(pol(\sigma))$ is visible to $A$. The case when $\mathbf{\Sigma} = \emptyset$ is trivial so we concentrate on the other case. We then have

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \{\sigma(\vec{\mathbf{a}})\} \cup \mathbf{\Sigma} \textbf{ do } c'], S\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})}_A \langle \mathbb{E}[c'; \textbf{for } \sigma(\vec{a}) \textbf{ in } \mathbf{\Sigma} \textbf{ do } c'], S[\vec{a} \mapsto \vec{\mathbf{a}}]\rangle$$

and

$$\langle \mathbb{E}[\textbf{for } \sigma(\vec{a}) \textbf{ in } \{\sigma(\vec{\mathbf{a}})\} \cup \mathbf{\Sigma} \textbf{ do } c'], T\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})}_A \langle \mathbb{E}[c'; \textbf{for } \sigma(\vec{a}) \textbf{ in } \mathbf{\Sigma} \textbf{ do } c'], T[\vec{a} \mapsto \vec{\mathbf{a}}]\rangle$$

and we are done.

That concludes the proof that well-typed programs are Paralocks secure.