UNIVERSITY OF SOUTHAMPTON

# Provenance in Distributed Systems

**A Process Algebraic Study of Provenance Management and its Role in Establishing Trust in Data Quality**

by

Issam Souilah

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Physical Sciences and Engineering
School of Electronics and Computer Science

April 2013

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Issam Souilah

We aim to develop a formal framework to reason about provenance in distributed systems. We take as our starting point an extension of the asynchronous $\pi$-calculus where processes are explicitly assigned principal identities. We enrich this basic setting with provenance annotated data, dynamic provenance tracking and dynamically checked trust policies. We give several examples to illustrate the use of the calculus in modelling systems where principals base their trust in the quality of data on the provenance information associated with it.

We consider the role of provenance in the calculus by relating the provenance tracking semantics to a plain one in which no provenance tracking or checking takes place. We further substantiate this by studying bisimulation-based behavioural equivalences for the plain and annotated versions of the calculus and contrasting the discriminating power of the equivalences obtained in each case. We also give a more denotational take on the semantics of the provenance calculus and look at notions of well-formedness and soundness for the provenance tracking semantics.

We consider two different extensions of the basic calculus. The first aims to alleviate the cost of run time provenance tracking and checking by defining a static type system which guarantees that in well-typed systems principals always receive data with provenance that matches their requirements. The second extension looks at the ramifications of provenance tracking on privacy and security policies and consists of extending the calculus with a notion we call filters. This gives principals the ability to assign different views of the provenance of a given value to different principals, thus allowing for the selective disclosure of provenance information. We study behavioural equivalences for this extension of the calculus, paying particular attention to the set of principals composing the observer and its role in discriminating between systems.

# Contents

# List of Figures

# Declaration of Authorship

I, Issam Souilah, declare that the thesis entitled "Provenance in Distributed Systems : A Process Algebraic Study of Provenance Management and its Role in Establishing Trust in Data Quality" and the work presented in it are my own, I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published in:
  - Issam Souilah, Adrian Francalanza, and Vladimiro Sassone. A formal model of provenance in distributed systems. In James Cheney, editor, *TaPP*. Usenix, 2009.

Signed: ______________________________

Date : ______________________________

# Acknowledgements

Perhaps one day, computers will provide the sound and complete provenance of a PhD thesis. Until that day arrives, I can only offer my apologies to anyone I may miss in the following.

Thanks to my supervisor Prof. Vladimiro Sassone for his support, guidance and patience.

Thanks to Dr. Adrian Francalanza for his contributions to this research.

Thanks to Dr. Julian Rathke for examining this work and offering his insights and ideas.

Thanks to everyone in the Dependable Systems and Software Engineering group.

Thanks to all my friends; you have been a welcome distraction.

Thanks to my parents, my brothers and my sister.

Thanks to my wonderful wife Salima.

*To my grandparents*

# Chapter 1

# Introduction

*Ubiquitous (*or *Pervasive) Computing* is a vision of a future, that is drawing ever so closer, where computational entities, both software and hardware, become an integral part of the environment in which we conduct our daily activities. In this vision, first made popular by Mark Weiser [70], computers move from the desktop to public spaces, as they once moved from the mainframe to the desktop, and become embedded in everyday life, almost vanishing completely as they blend in with their background. The entities that will inhabit the ubiquitous computing space are envisioned to be autonomous, mobile and context-aware. The smartphones and tablets of today, benefiting from a multitude of connectivity options and equipped with a wide range of sensors, can already be seen to exhibit a higher degree of these characteristics than has ever before been associated with electronic devices. They are computers that have eschewed the rigidity of our desktops, choosing instead to be a companion in the journey rather than a stop along the itinerary. Their most important contribution is in making the world's information, still largely unorganised and not universally accessible and useful, at least constantly available. Not only that, but in their simplicity they turned every reader into a writer, and blurred the lines between the reporter, the photographer, the film maker, and their audiences. The world's information is increasing at an unprecedented rate. The platform is a bazaar.

There was a time when people complained about lack of information; that time has long gone. The challenge today is that of choice. Faced with a plethora of information, and of information *sources*, what do we choose? The answer used to be in the information itself; choose information that is accurate, current, relevant, and in general information that exhibits a myriad of other properties deemed essential for a particular purpose. That answer is still correct. However, what has changed, and what is an intrinsic feature of ubiquitous computing, is the high level of uncertainty. In all but the simplest cases, it is difficult, expensive, or outright impossible to tell how accurate, current or relevant a piece of information is. In these settings, *computational trust* has been proposed as an alternative, or more realistically a complementary, method of judging the quality of information. Inspired by the principles of trust that govern human interaction, computational trust moves the answer from the information to its sources, its *provenance*. The premise is as follows, coupled with an underlying web of trust, the provenance of information should give us a fairly good indication of its quality. It is a simple premise. Good sources produce good information.

The ubiquitous computing vision has been a major aspiration, responsible for driving research in computer science and related fields for the last two decades. The challenges it poses, scientific, engineering and social, reflect the huge scale of ubiquitous computing. Tackling these challenges would allow us to understand the entities of ubiquitous computing, build them to meet our ever more complex requirements, and study their impact on society. The scientific challenge, being arguably the most fundamental of the three, constitutes the basis for the UK Grand Challenge on Science for Global Ubiquitous Computing [41]. Its aim is to obtain a theory on which to ground the analysis and understanding of computational entities built in the ubiquitous computing setting. Ultimately, such a theory would give us the tools we need to be able to build ubiquitous computing systems whose behaviour we can rigorously prove to be correct. We expect this theory to have a number of facets [47], each corresponding to a different feature of ubiquitous computing and addressing the problems stemming from it. In this thesis, we concern ourselves with one such feature, distribution, and one such problem, provenance. In this thesis, we concern ourselves with provenance-based trust in distributed systems.

In this introductory chapter, we give an overview of the thesis and explain its context, scope and main contributions. We split the chapter into five sections, entitled what, why, how, who and where. The choice of these section titles is an allusion to names of provenance notions commonly used by the provenance community. However, more than serving just as an allusion, these section titles do in fact reflect fundamental questions about the work presented in this thesis. In a way, these sections give the provenance of the thesis. In Section 1.1, *what*, we define the main topic and context of the thesis and proceed to describe the research problems that our work attempts to tackle. Then, in Section 1.2, entitled *why*, we give our motivations for studying provenance and the reasons we think this work is worthwhile. After that, in Section 1.3, we describe our approach for studying provenance and review the main tools and techniques we plan to use to answer our questions. The aim of Section 1.4, *who*, is to define the target audience of the thesis, which it does by making explicit our assumptions about what the reader is expected to already know and describing the prerequisite background knowledge. We conclude the chapter with Section 1.5 where we review our contributions, give an overview of the different versions of the calculus and outline the structure of the thesis, listing *where* each of the main research items is discussed.

## 1.1 What

### 1.1.1 Provenance

As a concept, provenance is synonymous with source, origin and derivation and may also mean the history of the ownership of a work of art or an antique, used as a guide to *authenticity* or *quality* [59]. More concretely, provenance refers to a *documented record* of such source or history, and depending on the particular setting, this record may include information about the influences, contributions as well as other *historical* or *contextual* information which may be deemed relevant and useful. Motivated by its potential uses in various computing settings, provenance of data and other digital artefacts is garnering increased interest and is emerging as an important research topic [17]. Indeed, provenance is valuable for many applications such as auditing, detecting errors and ensuring reproducibility of results of experiments. It is also central to the *trust* one places in data, since it can be used as an indicator of *quality*, especially in settings where the verification of other attributes of data may not be possible.

### 1.1.2 Provenance tracking

Provenance tracking is the problem of recording provenance information, and has been studied in a wide range of settings, including databases [8, 9, 15, 19, 28], scientific computing [30, 31, 60, 67] and file systems [56]. Initially, most of the work on provenance defined what provenance means based on intuitive and informal concepts such as *influences*, *contributes to*, and *depends on*. These notions were then used to provide mostly ad hoc implementations, with no formal guarantees as to their correctness or adequacy. However, lately there has been a surge of interest in underpinning the more theoretical principles of provenance [8, 15, 28]. These theoretical works aim to establish a mathematical and semantic basis for provenance, which is important if we are to compare different notions of provenance or assess the correctness of their implementations. The present work falls in this latter line of research and aims to provide a formal study of provenance-based trust in concurrent and distributed systems.

### 1.1.3 Distributed systems

We aim to study provenance in settings similar to those of ubiquitous computing. We believe the distributed nature of those systems is one of their most defining features. Therefore, we concern ourselves in this work with provenance in distributed systems. We model these latter as sets of principals or agents, possibly belonging to different trust domains, that communicate by message passing. As information may cross the boundaries of trust domains, principals need a way of judging its quality. We advocate the use of provenance for this, and hence, the notions of provenance we are interested in studying are those that could form the basis for trust in the quality of information in these systems.

### 1.1.4 Research questions

The aim of the previous three sections, and indeed of all of the chapter so far, was to define the basic concepts behind our work and to give a general idea of its context and scope. We now give a more thorough description of the research questions we aim to tackle in the rest of this thesis.

At a high level, we aim to define notions of provenance suitable for use as a basis for trust in distributed systems. We think of trust as primarily a relationship between principals and of provenance as a means of extending this relationship to data. More specifically, the provenance of data should allow a principal to use its beliefs about the trustworthiness of other principals to determine the trustworthiness of data produced, transmitted or otherwise affected by these principals. Trust beliefs between principals and how they are used to determine trust in data are local to each principal and are likely to depend on the domain being modelled and on the intended use of data. Therefore, it is not the role of provenance, and therefore neither of this work, to prescribe any particular way of deriving the trust associated with data from the trust associated with the principals appearing in its provenance. Instead, provenance should be merely informative, documenting the role of each principal in transmitting a piece of data from its original producer to its current owner, and therefore enabling each principal to judge the trustworthiness of data according to its own trust policies. We aim to look at different approaches for tracking such provenance information and to study the properties of both the provenance notion itself, as well as those of the different provenance tracking techniques. A natural way to track provenance information is by instrumenting the run time environment so that it annotates every piece of data with its provenance and updates these provenance annotations whenever an operation is performed on the data. Since such a modification of the run time environment is likely to have performance ramifications, conservative approximations of provenance that can be achieved by static analysis are desirable and therefore will investigated in this work. At least for the purposes of this work, we think of provenance management as a feature that would be added on top of the existing infrastructure of a distributed environment, no doubt introducing several ramifications. We already mentioned the performance ramification and the need for static provenance tracking. Perhaps more important, the disclosure of provenance is likely to pose privacy and security concerns and therefore principals should be able to control who has access to provenance pertaining to their actions. We plan to address this by studying programming constructs that allow principals to control the disclosure of provenance information.

The approach we take in this work is theoretical and is carried out in a process algebraic setting. This latter gives us all the basic tools and techniques we need to be able to model and reason about distributed systems, and allows us to build on the large body of readily available results by the process algebraic community. A more detailed description of our approach is given in Section 1.3.

In the following, we list the questions we aim to answer in the thesis:

1. What is provenance? Here, we only concern ourselves with what provenance means in the context of distributed systems, and mainly as an enabler for judging the quality of data. The provenance of a value should contain *enough* information about the role of each principal in getting that value to its current state. In particular:

    - Originator: the principal that produced the value matters: the trust policy of a principal might require it to only consume (or not consume) data that originated from a particular source.
    - Transmitter: the principals that transmitted the value matter: the trust policy of a principal might require it to only consume (or not consume) data that was transmitted by a select set of principals.
    - Enabler: the principals that enabled the transmission of the value matter: the trust policy of a principal might require it to only consume (or not consume) data that was transmitted on communication channels supplied by a select set of principals.
    - Order: the order of events matters: the trust policy of a principal might require it to only consume (or not consume) data that was transmitted between a select set of principals and in a particular order.

- Type: the type of event matters: the trust policy of a principal might require it to only consume (or not consume) data that was sent but not received (or vice versa) on a channel supplied by a select set of principals.

2. How do we track provenance? Having answered the previous question and defined a suitable notion of provenance, the next natural question to ask is how do we plan to track this provenance information. We already mentioned two possible ways of tracking provenance information; at run time by annotating each piece of data with its provenance and updating these annotations after each relevant computational step, and at compile time by using a static analysis to predict and approximate the run time provenance of data. These two provenance tracking methods will differ in their properties, and we wish to compare them based on three main criteria:

    (a) Soundness: does provenance tracking include any information that is incorrect with respect to the past of the data?

    (b) Completeness: does provenance tracking exclude any information that is deemed essential by the five criteria of the previous section?

    (c) Efficiency: does provenance tracking cause any performance overhead?

3. How do principals use provenance? The aim of tracking provenance in a distributed system is to make it available to principals in order to enable them to make decisions about how to use the data they receive from other principals, especially when those principals belong to other trust domains. Hence, the task of making that provenance available to principals in an accessible and useful way is at least as important as the task of tracking provenance itself. The main requirement here is that principals are able to express trust policies that depend on all five criteria of provenance.

4. What are the properties of the provenance notions? Studying the properties of a provenance notion allows us to better understand it and compare it to other provenance notions. This in turn makes it possible for us to decide which provenance notion to use in a particular environment or for a particular application.

5. What are the properties of the provenance tracking techniques? In addition to the above, since there may be multiple approaches to tracking provenance information for a particular provenance notion, it is important that we compare and contrast these different provenance tracking approaches to be able to tell which approach is suitable for each of the circumstances that we are interested in.

6. What is the impact of adding provenance tracking to a given computer system? The addition of provenance tracking to a computer system is likely to have many

ramifications, the most important of which is probably around the privacy and security requirements of this system. Therefore, a provenance management system should give principals full control over the disclosure of provenance information relating to their actions.

## 1.2 Why

### 1.2.1 Quality of information in the bazaar

The environments envisaged by ubiquitous computing are characterised by vast numbers of computational agents interacting to create, transform and publish data. This interaction makes use of, and produces, large *markets of data* whose quality, reflected in several criteria including for example accuracy, currency, and relevance, varies immensely. These data markets are open to agents belonging to different trust domains. Each agent is equipped with a *trust policy* arising from its local beliefs about the trustworthiness of other agents and governing its interaction with them. An agent's beliefs also affect the weights it ascribes to the different dimensions of data quality. Moreover, an agent is likely to perceive the quality of a piece of data differently depending on its intended use. This latter idea may be seen as corresponding more to the intuitive notion of the suitability of data for a particular task than the more general notion of data quality. Indeed, while a piece of data that is perceived to be incomplete may be suitable for some non-critical applications, it would be regarded unsuitable by most agents for applications with a critical nature. This distinction is irrelevant for the aims of the present work however as we do not, nor do we need to, address the problem of directly defining or measuring the quality of data. Instead, we aim to provide programmatic tools to allow agents to implement provenance-based trust policies and study the formal properties of such systems. The beliefs of agents and their perceptions of what constitutes data of high quality are implicit in the implementation of their trust policies.

### 1.2.2 The role of provenance

The quality of data may be judged relative to the local beliefs of an agent, the intended use of the data and its different attributes. However, it is often the case that information about the latter may not be available or may be impossible to verify independently. In these cases, information about the role each agent played in arriving at a piece of data, that is its provenance, may be used as a "metric" in determining the trust an agent should place in its quality. In fact, such use of provenance information is familiar from many real life situations. As an example, consider our use of brand names when deciding what products to buy; actually for many people this yields more influence than any other, sometimes more important, attribute in determining whether to buy a particular product or not. Often, this use of provenance goes even further where it is not only the maker of a particular product that influences the decision of what to buy, but also the retailer, the makers of the different components, and any other organisation or entity involved in creating, assembling, selling, promoting or marketing the product. Such use of provenance is also common in other applications and is usually guided by simple intuitions regarding the credibility, authority and honesty of the agents we interact with.

## 1.3 How

In this section, we describe our approach and the tools and techniques we plan to use to answer the questions we posed in Section 1.1.

### 1.3.1 Process calculi

We are interested in the formal foundations of provenance in distributed systems. Therefore, we need a way of formally modelling and reasoning about these systems. To that end, we make use of process calculi. Process calculi have matured for over three decades to become the standard formalism for studying concurrent and distributed systems. Their success does not only lie in their use as a language for modeling concurrent systems and studying their properties, but also, and arguably even more crucially, in providing a hotbed for the development of a wide range of techniques and tools, both theoretical and practical.

In this thesis, we start with a variant of the asynchronous $\pi$-calculus [6, 34]. The $\pi$-calculus, introduced by Milner, Parrow and Walker [48, 49], provides an elegant language for describing systems whose components interact by communication and whose network topology may change over time as components acquire new communication links or channels. The asynchronous $\pi$-calculus restricts communication to the asynchronous case. We find this to be a more faithful model of distributed systems, especially for the type of settings that motivate our work. We also make use of some of the theoretical frameworks developed around process calculi, the most important of which is probably bisimulation as we describe later in this section.

The asynchronous $\pi$-calculus gives us all we need to model and analyse distributed systems. However, the notions of provenance we are interested in studying rely on being able to model agents belonging to different trust domains. To do this, we extend the asynchronous $\pi$-calculus with principal identities or locations. What this gives us is a calculus where the trust domain that a process belongs to is made explicit, and the communication of messages between processes is allowed to cross trust boundaries. The role of provenance in judging the quality of data then becomes clearer. The basic model is described in detail in Section 3.1, and its extension with provenance is studied in Section 3.2.

### 1.3.2 Bisimulation

Bisimulation is arguably the most important contribution of Concurrency Theory to Computer Science [65]. One of many process equivalences to be proposed, bisimulation is distinguished by its coinductive definition and associated proof technique. Intuitively, two processes are related by bisimulation, or said to be *bisimilar*, if they can match each other's actions, *ad infinitum*.

In this work, we use bisimulation for two reasons. First it helps us understand the role of provenance tracking in the calculus by comparing the equivalence we obtain in the plain calculus with the one we get when we augment the calculus with provenance. Secondly, when we look at the security ramifications of provenance tracking in Chapter 7 and Chapter 8, process equivalence and bisimulation help us specify some of the privacy and security goals we wish a given system to achieve.

### 1.3.3 Static analysis

Static analysis refers to the analysis of computer programs without executing them, usually performed on the source code. The aim of static analysis is to catch and fix bugs at compile time to avoid errors at run time when their cost may be more expensive, ranging from minor annoyance and lost productivity to critical safety and security breaches. Static typing, probably the most popular and successful form of static analysis, consists usually of annotating a program with type information and verifying that certain properties would never get violated during the execution of the program. Static typing may be contrasted with dynamic typing where checks are performed at run time to ensure the program never violates the desired properties. The advantage of static typing over dynamic typing is the elimination of run time checks and their performance overheard.

In this work, we use static type checking to implement a form of provenance tracking that does away with run time provenance checks. This provides a conservative approximation of provenance that avoids the performance overhead of dynamic provenance checking and which might be desirable in certain cases. In addition to this, it sheds more light on the properties of the provenance notion we study in this work. Static provenance tracking is studied in Chapter 6.

## 1.4 Who

### 1.4.1 Prerequisites

As we have already mentioned, we are interested in understanding the formal underpinnings of provenance in distributed systems. Our approach in accomplishing this draws on the large repertoire of tools and techniques developed by the theoretical computer science community. These include programming language semantics, logic, type systems, process calculi and bisimulation. We explain the intuition behind each concept as we use it, however a comprehensive exposition of the whole background is naturally beyond the scope of this thesis. The reader is referred to Nielson and Nielson [57] for programing language semantics, Huth and Ryan [37] for logic, Pierce [61, 62] for type systems, Milner [46] and Sangiorgi [66] for the $\pi$-calculus, and Sangiorgi [65] for bisimulation.

## 1.5 Where

We split this section into three parts. In the first part, we offer an assessment of the contributions of this thesis, using the questions described in Section 1.1.4 as our baseline. The second part provides an overview of the different versions of the calculus introduced in this thesis. In the third and final part, we give a brief rundown of the content of the thesis.

### 1.5.1 Contributions

The thesis makes the following four main contributions.

#### 1.5.1.1 The provenance calculus

Our first contribution is the definition of the provenance calculus in Chapter 3. By presenting the calculus in two versions, we make explicit the extensions that are needed to incorporate provenance into the calculus. More precisely, three main extensions are made to the basic process calculus used as our starting point. The extensions are as follows.

**Provenance annotated data.** All data exchanged between principals is annotated with meta-data representing its provenance. These provenance annotations take the form of nested sequences of events, mirroring the send and receive actions of the calculus and recording the role each principal plays in routing a value to its destination. These provenance sequences meet all five requirements of provenance set out in Section 1.1.4:

- Originator: the principal that produced the value can be found as the author of the first (right-most) event in the sequence. For example, in the sequence $c!\kappa_{c1}$ ; $c?\kappa_{c2}$ ; $b!\kappa_{b1}$ ; $b?\kappa_{b2}$ ; $a!\kappa_{a1}$, principal $a$ represents the original producer of the value.
- Transmitter: all principals that transmitted the value can be found as the authors of events in the provenance sequence. For example, in the sequence $c!\kappa_{c1}$ ; $c?\kappa_{c2}$ ; $b!\kappa_{b1}$ ; $b?\kappa_{b2}$ ; $a!\kappa_{a1}$, the value was produced by $a$ and subsequently transmitted by $b$ and $c$ to its current location.
- Enabler: the principals that enabled the transmission of the value can be found in the provenance of the channels used. For example, in the sequence $c!\kappa_{c1}$ ; $c?\kappa_{c2}$ ; $b!\kappa_{b1}$ ; $b?\kappa_{b2}$ ; $a!\kappa_{a1}$, the provenance subsequences $\kappa_{c1}$, $\kappa_{c2}$, $\kappa_{b1}$, $\kappa_{b2}$, and $\kappa_{a1}$ all give the provenance of the channels used for communication and therefore information about the principals that enabled the value to be communicated to its current location.
- Order: provenance sequences are ordered from right to left. The right-most event is the oldest while the left-most event is the most recent.
- Type: provenance sequences may contain two types of events; send events of the form $a!\kappa$ and receive events of the form $a?\kappa$. These mirror the two types of communication actions available in the calculus.

**Provenance tracking semantics.** The provenance annotations of values are kept up-to-date as the system evolves by instrumenting the operational semantics of the calculus. This ensures that each computational step of the system results in updating the provenance of relevant values as required. The provenance tracking semantics is sound and complete in that it records information which is correct and which meets all five criteria of the provenance notion. However, it poses both space and time overhead; it records provenance in the form of annotations which require extra space to store, and the recording of provenance takes place at run time posing performance overhead.

**Dynamic provenance checking.** Principals make use of provenance by specifying provenance policies. These are patterns against which the provenance of values is checked automatically by the operational semantics of the calculus to guarantee that principals only consume data with provenance that meets their requirements. Patterns allow principals to express conditions that depend on all features of provenance, meeting the third requirement of Section 1.1.4.

In summary, the above three features answer the first three questions of Section 1.1.4. The provenance notion we propose is aimed to capture the role of each principal in getting a value from its original source to its current state. This meets all five requirements of provenance. Our first answer to the question of how to track provenance takes the form of automatically updating the provenance annotations with new information as the computation proceeds. Although it poses extra performance overhead, this method of tracking provenance is sound and complete. Principals are able to express trust policies that depend on the full range of information available in provenance annotations, and to check these provenance annotations against their local trust policies, hence making fully informed decisions on whether to consume the data or not. The local trust policies of principals are represented as patterns in a suitable pattern language. Principals associate policies with their input channels to indicate what provenance is required for each channel on which they consume data.

#### 1.5.1.2 Properties of provenance

We study the properties of the provenance calculus in two different ways. The first, covered in Chapter 4, consists of defining two behavioural equivalences for the plain and annotated versions of the calculus and contrasting their discriminating power. The second is explored in Chapter 5 and takes a denotational approach to look at two properties of provenance. The simpler property is known as *well-formedness* and characterises those provenance sequences that are possible, that is that could arise in a valid computation. The second property, *correctness*, is based on defining the semantics of provenance sequences as formulae in a specially devised logic. It characterises when a provenance sequence could be said to be a truthful record of the past.

These constitute our answers to the fourth and fifth questions of Section 1.1.4. By contrasting and comparing the two versions of the calculus, we are able to highlight the properties of provenance and its role in the calculus. Well-formedness provides a syntactic definition of what provenance sequences "make sense". Provenance sequences deemed ill-formed would never arise in a computation based on the rules of our calculus. The soundness property looks at what provenance sequences are meant to represent: records of some past behaviour of the system. As such, a provenance sequence is sound only if the information it conveys about the past is true.

#### 1.5.1.3 Static provenance tracking

The provenance policies of principals are enforced dynamically by the operational semantics which checks the provenance of values against the policies of principals before permitting the principals to consume the values. This causes extra performance overhead at run time. To remedy this, we propose a type inference system in Chapter 6 that approximates the provenance of values at compile time and guarantees that in well-typed systems principals always consume data with provenance that meets their requirements.

This is aimed as an alternative way of keeping track of provenance information and hence forms part of the answer to our second question, how do we track provenance? The static analysis eliminates the performance overhead of run time provenance tracking. It provides a conservative approximation of the provenance of values, guaranteeing both soundness and completeness of provenance. However, being a conservative approximation, it does rule out certain input transitions that would be allowed by run time provenance tracking. Chapter 6 gives a thorough comparison of the two provenance tracking approaches.

#### 1.5.1.4 Provenance security

The provenance notion we study in this thesis records the role that each principal played in getting a given value to the state it is in. What this does is reveal to any principal that consumes the value some of the actions that these principals have been involved in. However, this might be against the security policies of these principals. In Chapter 7, we propose an extension to the calculus that gives principals the ability to control how much of the provenance of the values they publish is visible to other principals. We study the properties of this extension of the calculus in Chapter 8, where we propose a novel notion of behavioural equivalence that enables us to analyse how much provenance different sets of principals are able to see.

As we already mentioned in the final question of Section 1.1.4, security is one of the main concerns when tracking provenance and hence it is the one we devote Chapter 7 and Chapter 8 to studying. The notion of filters we propose in Chapter 7 recognises that ownership of provenance should rest with the principal whose actions the provenance describes. Filters give principals full control over all provenance information pertaining to their own actions. This guarantees that provenance tracking does not violate the privacy and security policies of principals.

### 1.5.2 Versions of the calculus

We study provenance in the context of a process calculus. We start with a simple extension of the asynchronous $\pi$-calculus and incrementally build on it to introduce various features for provenance management. In particular, there are three main versions of the calculus and they are as follows:

- The plain version: forms the computational core on top of which the other versions add provenance management features.
- The annotated version: extends the plain version with provenance annotated data, dynamic provenance tracking and dynamically checked provenance policies.
- The filtered version: adds filters to the annotated version in order to enable principals to control the disclosure of provenance.

We denote the structural congruence relation of the plain version with $\equiv_p$, its reduction relation with $\rightarrow_p$ and its labelled transition relation with $\xrightarrow{\alpha}_p$. Variants of the these relations are also defined for the annotated and filtered versions of the calculus. They are denoted by $\equiv_a$, $\rightarrow_a$ and $\xrightarrow{\alpha}_a$ in the annotated version and by $\equiv_f$, $\rightarrow_f$ and $\xrightarrow{\alpha}_f$ in the filtered version. In general, functions, relations and any other definitions belonging to the plain version of the calculus will have the subscript $p$, those belonging to the annotated version will have the subscript $a$, and those belonging to the filtered version will have the subscript $f$. We also introduce several minor variants of the calculus throughout the thesis. We explain the role of each variant as we introduce it.

### 1.5.3 Structure of the thesis

The remainder of this thesis consists of eight chapters. Chapter 2 reviews related work on provenance. The following three chapters, that is chapters 3, 4 and 5, concern the basic formalism. They introduce the provenance calculus and study its properties. Chapter 6 looks at static enforcement of provenance policies while chapters 7 and 8 aim to address the security ramifications of provenance tracking by proposing an extension to the calculus with security primitives. Chapter 9 concludes the thesis. In the following, we comment on each chapter in more detail.

In Chapter 2, we provide a review of the provenance literature. We start by looking at simple forms of provenance. We think of these as precursors to current research on provenance. We then move on to more recent research efforts. We cover some of the main lines of provenance research in the database and the scientific workflow communities. We also look at cross-cutting concerns, namely formalisation, security and standardisation.

In Chapter 3, we give an overview of the calculus. We start with the plain version with no provenance annotations and then proceed to describe how provenance annotations, tracking and querying are added to this basic setting. Provenance querying is performed using pattern matching, the details of which are orthogonal to our aims and hence are left unspecified at this stage. However, in order to give concrete examples, we define a sample pattern matching language based on regular expressions. We use this to give several examples to illustrate the features of the calculus.

Chapter 4 studies the properties of pattern matching languages. It then compares the plain and annotated versions of the calculus, firstly by contrasting their reduction relations and secondly by using behavioural equivalences based on barbed congruence. By contrasting the equivalences obtained for each version, we highlight better the role that provenance tracking plays in the calculus.

In Chapter 5, we consider the semantics of provenance sequences and study properties of our provenance tracking reduction relation. We define a temporal logic aimed to serve as a target for defining the denotation of provenance sequences. We then use this to formulate and prove a property we call provenance correctness. This latter is aimed to capture the intuitive idea that what the provenance information tells us about the past of values does indeed coincide with what actually took place.

In Chapter 6, we look at how to eliminate the run time overhead caused by dynamic provenance tracking and checking. We do this by using a type and effect inference system which gives a conservative approximation of the provenance of values and guarantees that principals always consume data with provenance that matches their policies.

Chapter 7 considers security issues raised by provenance tracking and proposes an extension to the calculus to enable each principal to control the disclosure of their provenance information to other principals. The mechanism proposed is based on the idea of filters and gives principals total ownership of provenance information pertaining to their own activities. Access and control of provenance information belonging to other principals is, however, at the discretion of the authors of that provenance.

In Chapter 8, we study properties of the extended calculus and the security guarantees offered by filters. This is done by defining a notion of behavioural equivalence that takes the set of principals that are available to the observer into account. Varying the set of principals available to the observer gives us equivalences with different discriminating power and allows us to highlight the views of provenance that are associated with different principals.

In Chapter 9, we review our contributions and suggest possible avenues for future work.

# Chapter 2

# The History of Provenance

Provenance, in one form or another, has always been an integral part of computer systems. Indeed, a wide range of familiar applications such as file systems, revision control systems, and backup systems keep track of different types of information recording their execution and past states. Although they may refer to these types of information under a variety of other names, such as logs, audit trails or history trees, all these types of information are really just different *notions* of provenance. It is important to note here that although essential to the functioning of these and many other systems, until relatively recently, provenance was not thought of as an important problem in its own right. Instead, it was intertwined with the many other features those applications provided, and sometimes like in the case of logs kept by software applications, it came only as an afterthought. This has meant that lessons learned from the development of one notion of provenance could not benefit the development of another, and more importantly, that the guarantees offered by a particular notion of provenance were not at all clear.

It was not until around 2000 that provenance started to draw serious interest from the Computer Science research community. In fact, interest in provenance came simultaneously from several different sub-communities within Computer Science, including databases, scientific computing and the semantic web. Efforts to bring these communities together soon followed, and led to several events organised specifically for research on provenance. Moreau [51] notes that half the research papers on provenance were published in the last couple of years, most of them at these events. This is not only a testament to the success of these events, but also to the sizeable and growing provenance research community. There are a few reasons for this increase of interest in provenance research; the most important of which is probably the open and connected nature of the web. This has enabled the dissemination of information at an unprecedented rate and has meant that the premise of centrally-controlled, expertly-curated data repositories no longer holds. Instead, the reality we are increasingly dealing with is similar to that envisioned by ubiquitous computing, where disparate entities create, transform and publish data. Provenance, as we already mentioned, stands to play a major role in enabling us to judge the trustworthiness of data in such environments.

The remainder of this chapter is split into two main sections. In the first section, we look at the ubiquity of provenance in almost every type of computer system. We will note that many of these systems developed their own notions of provenance to meet their individual requirements. We will also note that for the most part, these notions of provenance were implemented in a rather ad hoc fashion and were never made separate from other features of these systems. In fact, these notions of provenance were not even thought of as instances of the same problem. After that, we move on to the second section where we focus on the recent rise of interest in provenance research. Here, provenance starts to be recognised as a separate important research problem in its own right, and the different research communities that work on provenance come together to look at unified definitions of provenance, shared and standardised representations of provenance and common approaches and strategies to implement provenance tracking systems. We review works on provenance in databases, in workflow and scientific computing as well as in the semantic web. We also review research on cross-cutting concerns such as provenance security and discuss recent standardisation efforts.

## 2.1 Prehistory

We use the term *prehistory* to refer, not to a particular span of time, but rather to a way of thinking about provenance. Prehistory is characterised by the lack of a clear separation between the provenance notions, the tools and techniques used to track provenance information, and the plethora of other problems that any nontrivial computer system is expected to deal with. Because of this, many of the systems belonging in this category, which we call *prehistoric provenance systems*, had to develop their own notions of provenance and their own implementations to track provenance information. The lessons learned from one system could not be shared with other systems, and this meant that prehistoric provenance systems have played very little role in advancing our understanding of provenance, even though some of them have been in existence for a very long time. It is worth noting that the problems with prehistoric provenance systems we mentioned correspond to two classical software engineering problems, that of the separation of specification (i.e. what is provenance) from implementation (i.e. how to track provenance), and that of the separation of concerns (i.e. separating provenance, security, performance, and other concerns from each other).

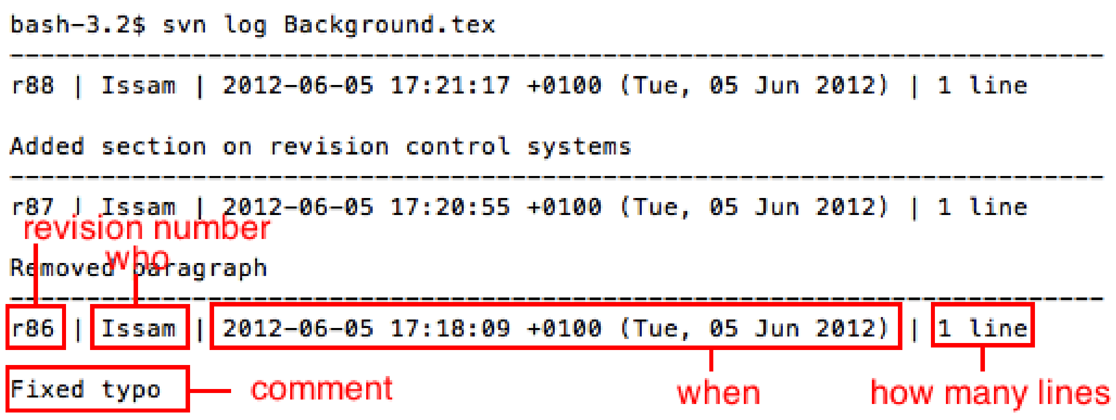
The list of individual systems that belong in this category is enormous. In fact, depending on where one draws the line of what information is considered provenance, it is possible to classify almost every computer system ever built as keeping some form of provenance information. An exhaustive review of every such system is clearly beyond the scope of this thesis, and probably too much for any single work to hope to cover thoroughly. What we do instead in this section is review a few particular classes of these systems. We think these highlight some of the most interesting notions of provenance found in familiar software applications. The classes of systems we look at include revision control systems, wikis, logging and the many other forms of history provided by applications as features to their users.

### 2.1.1 Revision control systems

Most widely used in software engineering, revision control systems help software developers manage changes to source code and other software development artifacts such as design documents, user manuals and configuration files. Software development is incremental by its nature, and in all active software projects, changes are constantly made to the code base as bugs are fixed and new features are added. To make this task manageable, to allow for accountability and traceability, and to enable large teams of developers to collaborate, revision control systems make use of a rich form of provenance tracking. Every change made to a file under revision control is tracked by the revision control system. Figure 2.1 shows sample history information kept by the revision control system Subversion [63] about a file. The figure shows summarised meta-data about each change, such as who made it, when it was made, how many lines were affected, and a brief description of the change supplied by the developer. More detailed information about the change, down to the exact state of the file before and after the change, can also be obtained from Subversion.

It is worth noting that revision control systems are not the preserve of software engineering and are actually used whenever there is a need to track and manage changes made to documents, especially in collaborative environments. As a result of this, revision control systems can also be found integrated in many other applications such as word processors, content management systems and wikis as we will see in the next section.

FIGURE 2.1: Provenance of a file in Subversion



```
bash-3.2$ svn log Background.tex
------------------------------------------------------------------------
r88 | Issam | 2012-06-05 17:21:17 +0100 (Tue, 05 Jun 2012) | 1 line

Added section on revision control systems
------------------------------------------------------------------------
r87 | Issam | 2012-06-05 17:20:55 +0100 (Tue, 05 Jun 2012) | 1 line

Removed paragraph
------------------------------------------------------------------------
r86 | Issam | 2012-06-05 17:18:09 +0100 (Tue, 05 Jun 2012) | 1 line

Fixed typo
```

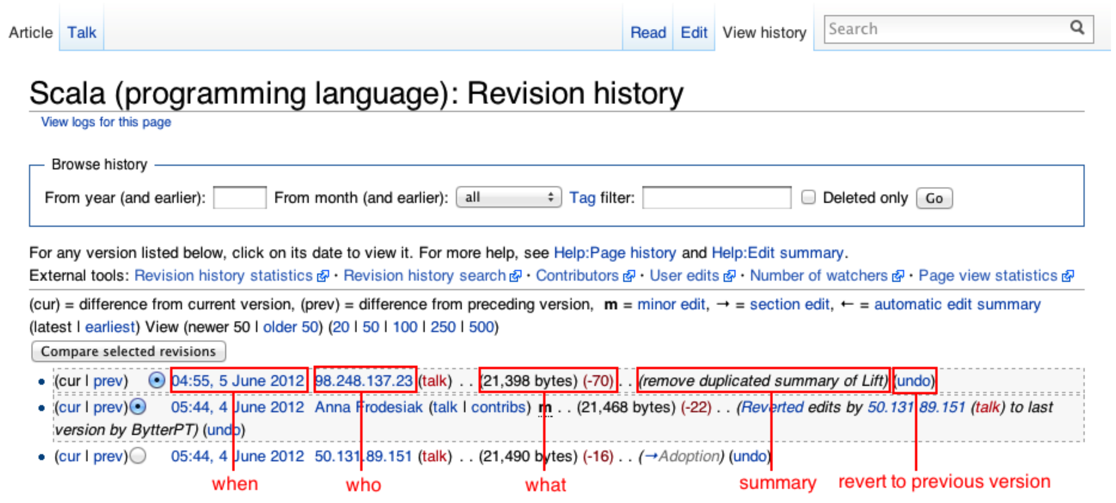
## 2.1.2 Wikis

Probably no single computer system or web site epitomises the democratisation of knowledge brought about by the rise of the web as much as Wikipedia.[1] Wikipedia is an online encyclopedia that employs the concept of a *wiki* [44], a website whose content can be edited by any user, to enable the collaborative creation of a high quality encyclopedia. The philosophy behind wikis, and Wikipedia in particular, is to make it as easy and quick as possible for users to add, edit and remove content. This is driven by the premise that more often than not, the changes made by users would be to correct existing mistakes rather than to make new ones, leading incrementally to better content. To that end, wikis try to forego traditional access control methods whenever possible in favour of a form of revision control that guarantees easy recovery from unacceptable content changes. As an example, Figure 2.2 shows an extract from the history of the Wikipedia page on the Scala programming language [58]. As can be seen from the figure, the revision history is quite similar to that provided by Subversion and shows every version of the page as a chronologically ordered list, going back to when the page was first created. Each line in the list represents a particular version of the page in question, and gives the following information about it:

- A timestamp showing when that version was created.
- The username or IP address of the user who created that version.
- How much of the content of the page was changed from the previous version, given both as the number of bytes changed and the number of lines added or removed.
- A brief textual summary describing what the change was.

[1]`http://www.wikipedia.com`

FIGURE 2.2: Revision history of a Wikipedia page



Scala (programming language): Revision history

View logs for this page

From year (and earlier): From month (and earlier): all Tag filter: Deleted only

For any version listed below, click on its date to view it. For more help, see Help:Page history and Help:Edit summary.
External tools: Revision history statistics · Revision history search · Contributors · User edits · Number of watchers · Page view statistics

(cur) = difference from current version, (prev) = difference from preceding version, **m** = minor edit, → = section edit, ← = automatic edit summary
(latest | earliest) View (newer 50 | older 50) (20 | 50 | 100 | 250 | 500)

- (cur | prev) 04:55, 5 June 2012 98.248.137.23 (talk) . . (21,398 bytes) (-70) . . *(remove duplicated summary of Lift)* (undo)
- (cur | prev) 05:44, 4 June 2012 Anna Frodesiak (talk | contribs) **m** . . (21,468 bytes) (-22) . . *(Reverted edits by 50.131.89.151 (talk) to last version by BytterPT)* (undo)
- (cur | prev) 05:44, 4 June 2012 50.131.89.151 (talk) . . (21,490 bytes) (-16) . . *(→Adoption)* (undo)

- A way to revert to the previous version of the web page if the change is deemed undesirable for whatever reason.

The revision history kept by the wiki software ensures that no change to a web page is ever lost. This is intended to protect the wiki against bad changes or outright vandalism as users are able to revert the web page to a previous good version whenever needed.

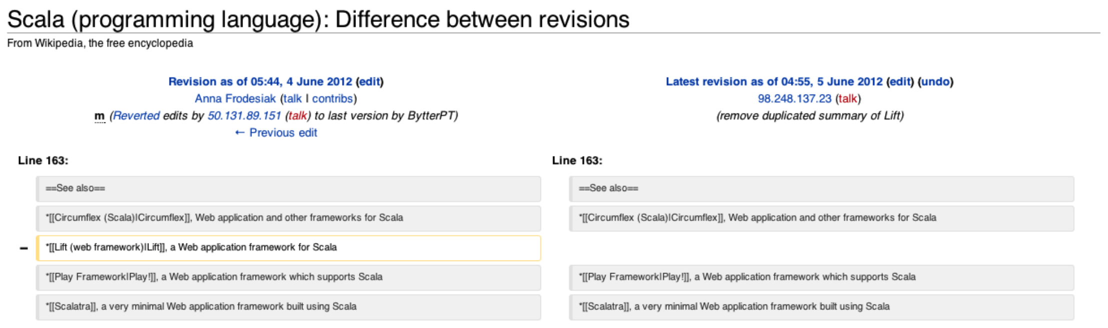
Revision control systems, both as used in software engineering as well as in wikis, can be seen as maintaining two types of provenance information:

- The different states (or revisions or versions) that a particular artifact, whether a source code file or a web page, goes through as it evolves. This allows users to compare two different revisions of an artifact to pinpoint where exactly they differ. Figure 2.3 shows the difference between two revisions of the Wikipedia page on Scala.
- Additional contextual information describing when each change took place, who made it as well as a short comment by the author of the change describing its intent and content. This is the information shown in Figure 2.2.

### 2.1.3 Logging

Another form of provenance that is ubiquitous in every software application is that kept in *logs*, persistent stores such as files or databases where applications record interesting events they encounter while running. That way it is possible to refer back to this information and use it for troubleshooting problems, analysing performance issues and

FIGURE 2.3: Comparing two revisions of a Wikipedia page



Scala (programming language): Difference between revisions

From Wikipedia, the free encyclopedia

| Revision as of 05:44, 4 June 2012 (edit)<br>Anna Frodesiak (talk I contribs)<br>m (Reverted edits by 50.131.89.151 (talk) to last version by BytterPT)<br>← Previous edit | Latest revision as of 04:55, 5 June 2012 (edit) (undo)<br>98.248.137.23 (talk)<br>(remove duplicated summary of Lift) |
|---|---|
| Line 163: | Line 163: |
| ==See also== | ==See also== |
| *[[Circumflex (Scala)ICircumflex]], Web application and other frameworks for Scala | *[[Circumflex (Scala)ICircumflex]], Web application and other frameworks for Scala |
| – *[[Lift (web framework)ILift]], a Web application framework for Scala | |
| *[[Play FrameworkIPlay!]], a Web application framework which supports Scala | *[[Play FrameworkIPlay!]], a Web application framework which supports Scala |
| *[[Scalatra]], a very minimal Web application framework built using Scala | *[[Scalatra]], a very minimal Web application framework built using Scala |

detecting intrusion. For example, the Apache HTTP Server[2] keeps two main log files, an error log file where it stores diagnostic information about the errors that it encounters and an access log file where it records all requests processed by the server. The exact content and format of these two log files is configurable by the server administrator, but a typical line in the access log file would record information about the request, such as the IP address of the client that sent it, the user that sent it, its date and time, its method and the resource requested. This would look something like this:

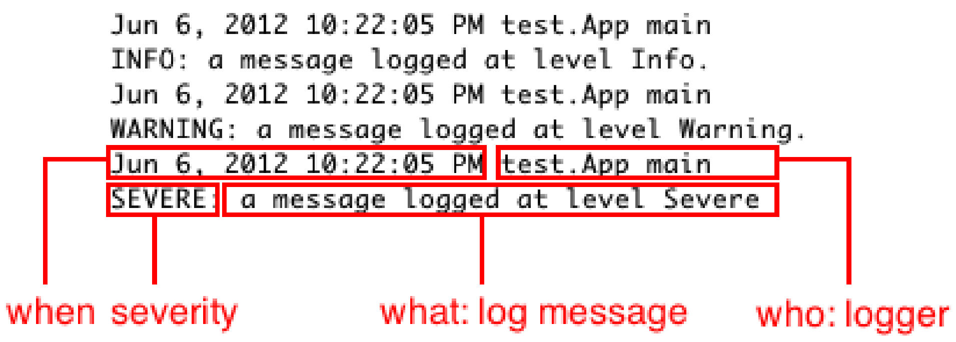
```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

In their most general form, application logs can contain any run time information a program cares to record. This is clearly too generic for any single provenance notion to adequately meet the logging requirements of every application. Hence, it is left as the responsibility of each application developer to define their own "provenance notion" and make sure their code records relevant and sufficient information in log files. Faced with what is arguably a very intractable problem, common efforts in logging have focused instead on developing tools and frameworks to manage the logging process. For example, programs written in the Java programming language benefit from a large selection of so called *logging frameworks*. Examples of these include the Java Logging API[3], Log4J[4] and Apache Commons Logging[5]. These frameworks aim to make it easier for developers to log run time information by splitting the task of logging into three components, each configurable separately:

- Logger: the job of a logger is to supply the other two components with the messages that need to be logged. It is the only component that application code needs to deal with directly. Application code calls methods of the logger with the messages or objects it would like to log.

[2]`http://httpd.apache.org/docs/1.3/logs.html`
[3]`http://docs.oracle.com/javase/7/docs/technotes/guides/logging/`
[4]`http://logging.apache.org/log4j/`
[5]`http://commons.apache.org/logging/`

FIGURE 2.4: Sample log messages using the Java Logging API



```
Jun 6, 2012 10:22:05 PM test.App main
INFO: a message logged at level Info.
Jun 6, 2012 10:22:05 PM test.App main
WARNING: a message logged at level Warning.
Jun 6, 2012 10:22:05 PM test.App main
SEVERE: a message logged at level Severe
```

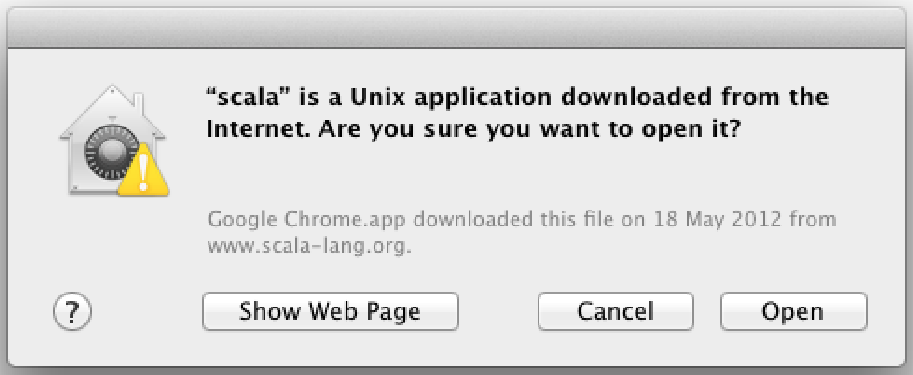
- Formatter: this is responsible for determining how to format log messages and aims to separate the presentation of log messages from their content.
- Appender: the appender receives the messages that need to be logged from the logger, uses the formatter to decide how these messages should be formatted, and then writes the formatted messages to the console, log file or any other output stream as required.

A logger is associated with a component, package or class in the application. It has a level which determines the severity of messages that should be logged. Most Java logging frameworks recognise some or all of the following logging levels: (1) FINEST, (2) FINER, (3) FINE, (4) CONFIG, (5) INFO, (6) WARNING, (7) SEVERE. Application code specifies the level each message should be logged at. Messages with the same level as the logger or higher would be sent to the appender; all other messages would be ignored. This allows developers to turn certain logging messages on and off as required. For example during development, in order to aid in debugging, all logging messages would be enabled by setting the logging level to the lowest, while when deploying the application in a production environment only higher levels are enabled to minimise the performance overhead of logging. Sample output from the default implementation of the Java Logging API is given in Figure 2.4.

### 2.1.4 History as a feature of software

Provenance is everywhere. Revision control systems, wikis, and logging are but a few examples. In fact, in addition to keeping log information, many applications have integrated some form of provenance as a feature for their users. For instance, the Mac OS X operating system warns users when opening files they downloaded from the Internet.

FIGURE 2.5: Provenance of files downloaded from the Internet in Mac OS X 10.7



**"scala" is a Unix application downloaded from the Internet. Are you sure you want to open it?**

Google Chrome.app downloaded this file on 18 May 2012 from www.scala-lang.org.

Figure 2.5 shows such a warning dialog box. It asks the user to confirm that they want to open the file or program, given its provenance. In this example, it says that the file the user is about to open is a Unix application that was downloaded by the Google Chrome web browser on 18 of May 2012 from the website `www.scala-lang.org`. This ensures users are aware of the source of files they open and aims to protect them from viruses and other malicious software.

Along similar lines, many applications keep track of recently opened items. For example, web browsers keep a record of recently visited web pages, word processors keep a record of recently opened documents, and Unix shells keep a record of recently executed commands. In fact, these ideas are made a lot more useful by the "undo" and "redo" features so common in many document editing applications. File systems are not an exception; they associate with each file meta-data recording contextual and historical information about the file such as when it was created and when it was last edited. Web cookies, stored by websites on a user's computer so that the user's activity on the website can be checked in future visits, are also a form of provenance. The list of applications and the list of forms of provenance is indeed immense. This section is only intended to give the reader an idea of some of the most common forms of provenance in software applications.

## 2.2 A bright new past

In the previous section, we looked at forms of provenance that have not been traditionally thought of as provenance. In a way, they are forms of provenance that have been referred to by many other names but provenance. It is interesting that we now recognise them as being just different, albeit sometimes very simple, notions of provenance. This realisation, and this common vocabulary, will hopefully have some positive impact on the way we implement provenance tracking in those systems.

This section aims to provide a brief review of the provenance literature. As noted in the introduction to this chapter, there has been a dramatic increase of interest in provenance over the last decade or so. The Computer Science research community looked at provenance in a variety of domains and for a variety of applications. For the sake of ease of presentation, we split the works into two topics: databases and workflows. We also group together works on cross-cutting concerns such as formal models, security and standardisation. This should make it easier for us to compare similar works and highlight common research themes. However, it is worth noting that there are works that cannot be neatly classed into one of the aforementioned topics. This is especially the case, perhaps, with the more recent research efforts that are the result of collaborations spanning multiple different fields within Computer Science, and sometimes even touching on the non-Computer Science aspects of provenance.

### 2.2.1 Databases

Probably the clearest area in provenance research is that of databases, and that is both in terms of classifying what works belong in this area and also in terms of the different provenance notions that people have proposed. In fact, the database research community categorises the different provenance notions applicable in databases simply in the form of the questions they answer. For example, one type of provenance in databases is known as *Where-provenance*, and as its name suggests, it tells us where in the input, a piece of data in the output of a query comes from. Another type of provenance is *Why-provenance* and it tells us why, in the form of tuples of data in the input, a tuple of data is present in the output of a query. The type of provenance known as *How-provenance* is the most complete of the three and tells us how a piece of data in the output of a query was derived from its inputs. Having said this, it is worth pointing out that the database research community has also defined other provenance notions that do not fit in general within the framework of Where, Why and How provenance. We comment on some of these at the end of the section.

#### 2.2.1.1 Statement of the problem

Provenance research in databases is concerned with identifying the origins of the data in the database and the process by which it arrived there [9]. An important instance of this problem is where the source of the data is another database. This setting is encountered often, for instance, in data integration, warehousing, views and in results of queries in general. It has many applications such as tracing the source of an error in the output of a query and keeping views up-to-date as their sources are updated. The main aim of this line of research is to relate *items in the output* of a query to *items in its input* such that the latter can be said to have *contributed* to the existence of the former. This can be rephrased as follows, given an output database $O$, which is the result of query $Q$ on input database $I$, that is we have $O = Q(I)$, the provenance of an item $o \in O$ are items $i \in I$ such that $i$ contributes to $o$. The exact definition of the provenance notion (i.e. what is meant by "contributes") as well as the granularity at which data in the input and output is considered (i.e. what is meant by "items" in the input or output) varies depending on the intended use of provenance. Hence, it is possible to obtain different notions of provenance by choosing different definitions of provenance or by looking at data at different granularities.

We already mentioned the three types of provenance that have been studied in the database literature: Where, Why and How. We take this grouping of provenance notions from Cheney et al. [16]. There, the authors review many of the provenance notions proposed by the database research community and group them under these three broad types. Their review includes the two original characterisations of Where and Why provenance introduced by Buneman et al. [9], the original definition of How provenance introduced by Green et al. [28], as well as other notions of provenance that can be grouped under one of these three types. We follow the same grouping when presenting the different notions of provenance in this section. We assume a relational model of data when giving examples. However, it should be noted that these notions are also applicable to other models of data, and most of the time the definitions carry over to those settings in a straightforward way.

#### 2.2.1.2 Where-provenance

The Where characterisation of provenance was introduced by Buneman et al. [9]. It is probably the most intuitive of the provenance notions in terms of definition. It takes as its granularity individual locations in the relation, that is it associates provenance information with the cells of a relation, and looks at how data is copied from the input of

**composer**

| name | dob | |
|---|---|---|
| J.S. Bach | 1685 | $c_1$ |
| G.F. Handel | 1685 | $c_2$ |
| W.A. Mozart | 1756 | $c_3$ |

**work**

| name | opus | title | |
|---|---|---|---|
| J.S. Bach | BMV82 | I have enough. | $w_1$ |
| J.S. Bach | BMV552 | NULL | $w_2$ |
| G.F Handel | HMV19 | Art thou troubled? | $w_3$ |

TABLE 2.1: Input relations

**result**

| name | dob | |
|---|---|---|
| J.S. Bach | 1685 | $r_1$ |
| G.F. Handel | 1685 | $r_2$ |

TABLE 2.2: Result of the first query

a query to its output. Given a piece of data in the output of a query, its Where-provenance is the piece of data in the input from which it was copied. This is best illustrated through an example, and for this we use one adapted from Buneman et al. [9]. Consider the two database relations `composer` and `work` in Figure 2.1. We use the labels $c_1$, $c_2$, ..., etc to refer to individual tuples within a database relation, and expressions such as $c_1(name)$ to refer to a particular data item in a tuple. So for example, $c_1$ refers to the tuple ("J. S. Bach", 1685) in the relation `composer` and $c_1(name)$ refers to the name attribute of that tuple.

Now let us consider the following SQL query:

```
SELECT c.name, c.dob
FROM composer c, work w
WHERE c.name = w.name
```

The output of this query with respect to the relations `composer` and `work` in Figure 2.1, is the two tuple relation `result` in Figure 2.2. Now take the tuple ("J. S. Bach", 1685) labelled as $r_1$ in the result of the query, and ask the question: where did the data item $r_1(dob)$, that is 1685, come from? The answer is that it was "copied" from the date of birth attribute of the tuple $c_1$ in the relation `composer`. Hence, we say that the Where-provenance of the data item $r_1(dob)$ in the result of our example query is the data item $c_1(dob)$ in the input relation `composer`.

### 2.2.1.3 Why-provenance

Why-provenance does not just look at where data was copied from, but also considers what other data in the input contributed to it. It associates with each tuple in the output, a subset of the input tuples, that is, it takes tuples as its granularity. The intuition behind it is rather simple, and as with Where-provenance, is best illustrated through an example. Consider the example query and relations we used for Where-provenance. We said that J. S. Bach's date of birth in the output, that is $r_1$(dob), was copied from the input location $c_1$( dob), and we took that to be its Where-provenance. Now, lets consider this data item again. It seems clear that its presence in the output of the query owes to more data items in the input than just $c_1$(dob). In fact, $r_1$(dob) would not have appeared in the result of the query if it wasn't for the tuple $c_1$ in the relation `composer`, and the tuples $w_1$ and $w_2$ in the relation `work`. More precisely, the example query joins the relations `composer` and `work` on their `name` attribute, and consequently, only composers whose names appear in both `composer` and `work` will appear in the result of the query.

Why-provenance traces its origins to one of the earliest works on provenance tracking in database management systems, and that is the work of Cui et al. [19] on the *lineage* model. The lineage model would associate $r_1$ in the result of our example query with the set $\{c_1, w_1, w_2\}$; this latter being *sufficient* to produce $r_1$ according to the query. A shortcoming of the lineage model, which can be seen from this simple example, is that it does not account for the fact that $w_1$ and $w_2$ are not both needed to produce $r_1$. Either of them together with $c_1$ would be sufficient to get $r_1$. This shortcoming was rectified by Buneman et al. [9] into the provenance notion that has become known since then as Why-provenance. Why-provenance accounts for the multiple ways in which a tuple in the output could be derived from the input according to a particular query. So for our example, the Why-provenance of $r_1$ is the set $\{\{c_1, w_1\}, \{c_1, w_2\}\}$, indicating that either $\{c_1, w_1\}$ or $\{c_1, w_2\}$ is sufficient to produce $r_1$.

#### 2.2.1.4 How-provenance

Why-provenance describes the different combinations of input tuples that contribute to producing a particular tuple in the output of a query. What it doesn't tell us however is *how* these tuples are used by the query to construct the output tuple. How-provenance, proposed by Green et al. [28], aims to do just that. It goes beyond Why-provenance by including additional structure in the provenance to represent how each tuple in the provenance was used. This additional structure is based on a semiring whose set is the set of tuple labels, and whose two binary operations represent join and union. Hence, the How-provenance of a tuple is a polynomial rather than a set of sets.

As an example of How-provenance, take the tuple $r_1$ in Figure 2.2. Its How-provenance is the polynomial $c_1 \cdot (w_1 + w_2)$. This tells us that $c_1$ joined with either $w_1$ or $w_2$ is sufficient

**result**

| name_1 | name_2 | dob | |
|---|---|---|---|
| J. S. Bach | J. S. Bach | 1685 | $r'_1$ |
| J. S. Bach | G. F. Handel | 1685 | $r'_2$ |
| G. F. Handel | J. S. Bach | 1685 | $r'_3$ |
| G. F. Handel | G. F. Handel | 1685 | $r'_4$ |
| W. A. Mozart | W. A. Mozart | 1756 | $r'_5$ |

TABLE 2.3: Result of the second query

to produce the tuple $r_1$ according to our example query. Now, compare this to the Why-provenance of $r_1$, which we said was the set $\{\{c_1, w_1\}, \{c_1, w_2\}\}$. The two are actually equivalent in this case, albeit they differ in format. In general however, Why-provenance can be derived from How-provenance, by simply dropping the additional structure and only retaining the set of sets of labels, whereas the inverse does not hold. To illustrate this, let us consider another example. Take the following query, which returns the names and dates of birth of composers who were born on the same day.

```
SELECT c1.name as name_1, c2.name as name_2, c1.dob as dob
FROM composer c1, composer c2
WHERE c1.dob = c2.dob
```

The result of the query with respect to the input relations in Figure 2.1 is the relation `result'` in Figure 2.3. The query performs a self-join on the relation `composer` to get the names of each two composers who share the same date of birth. Since this is a self-join, and the only condition in the query is that the date of birth is the same in the two instances of the relation `composer`, we get tuples such as ("J. S. Bach", "J. S. Bach", 1685), indicating the self-evident fact that J. S. Bach shares his birthday with himself. The Why-provenance and How-provenance of this tuple are $\{\{c_1\}\}$ and $c_1 \cdot c_1$ respectively. Here, the How-provenance of $r'_1$ tells us more than its Why-provenance. The How-provenance $c_1 \cdot c_1$ tells us that $c_1$ was joined with itself to produce $r'_1$. Contrast this with $\{\{c_1\}\}$ which only tells us that $c_1$ is sufficient to produce $r'_1$ but does not tell us that we actually need two copies of $c_1$ to get $r'_1$ from our example query.

#### 2.2.1.5 Discussion

**Comparison.** Each of the three provenance notions is natural and answers in an intuitive way to a question about the origin of data. The three questions, and three provenance notions, are clearly related and we have already talked about the relationship between Why-provenance and How-provenance. The relationship between the provenance notions is probably even more important to understand than the three individual notions as it sheds more light on the nature of these provenance notions, and on the nature of provenance in general. As we already mentioned, Why-provenance is contained in How-provenance. That is, How-provenance includes all the information that Why-provenance includes about the origins of a tuple, and adds even more information of its own. This additional information tells us how each tuple in the provenance was used by the query, and as we saw, this uncovers cases where a particular tuple is used more than once by the query for example. In a way, the relationship between Why and How provenance is easy to spot since they have the same granularity. Where-provenance, on the other hand, works at the finer granularity of individual data items or cells. Its relationship to Why-provenance was studied by Buneman et al. [9] where it was shown that, for data that is copied by the query from the input relations, the Where-provenance of a data item is contained in the Why-provenance of its tuple.

Comparing provenance notions as done above is based on their “expressiveness”. Another dimension by which provenance notions can be compared is that of invariance under query rewriting, or lack thereof. That is, does the provenance stay the same if we replace a query with an equivalent one? It turns out that none of the three provenance notions is invariant under query rewriting in general. This indicates that although all three provenance notions are meant to only capture the relationship between the output of a query and its inputs, part of the query itself “leaks” into the provenance. This means that the provenance of data according to these notions is in fact a relation between the input data, the output data *and* the query according to which the output data was derived from the input data.

**Applications.** Several applications of the provenance notions we looked at have been proposed and studied in the literature. For example, an interesting application of Where-provenance is in annotation propagation, copying meta-data from the input of a query to its output. In fact, this was studied in one of the very early works on provenance in databases by Wang and Madnick in their *Polygen model* [69]. Annotation propagation was also studied by Buneman et al. [10] and an implementation of it given in the DB-Notes annotation management system [5]. Green et al. [29] develop a prototype system, known as ORCHESTRA, to showcase an application of their How-provenance notion to the problem of propagating updates in a peer to peer data sharing network related by schema mappings.

**Other provenance notions.** The Where, Why, and How provenance models cover a lot of the provenance work that has taken place within the database community. However, not all the provenance notions that have been proposed fit within this model. For example, the Polygen model of Wang and Madnick [69] that we have already mentioned does not fit in any of the three provenance notions. Another provenance notion that does not fit is that proposed by Woodruff and Stonebraker [71], which generates lineage information by inverting processing operations rather than relying on meta-data. Benjelloun et al. [4] introduce an extension of relational databases they call ULDBs. Their extension integrates both uncertainty and provenance in one model, and this too is not an instance of any of the three provenance notions.

Research on annotated databases is also important for provenance. Annotated relations are not limited to provenance and appear in various other contexts and settings in the database literature such as in incomplete databases [39] and probabilistic databases [24, 42]. In these contexts, relations are extended to associate an annotation with each tuple. Now the problem that arises with these extended relations is how to treat the annotations when evaluating queries. There are two main approaches to this, *annotation propagation* where queries are applied to data and the annotations are merely propagated, and *annotation querying* where annotations are considered data and treated in the same way as other data by relational queries. Annotation propagation is usually considered an *implicit* treatment of annotations as annotations are propagated without any interference from the user, while annotation querying is more *explicit* as the user is required to manipulate the annotations as they see fit. In annotation propagation, relational algebra operators are adapted to perform something meaningful on the annotations (based on their semantics). Buneman et al. [8] study the expressiveness of implicit provenance propagation in query and update languages while Geerts and Bussche [25] give a comparison of the expressive power of the two approaches.

#### 2.2.1.6 Comparison with our work

At a high level, works on provenance in databases differ from our own in the granularity at which data is studied. In databases, provenance tracks the operations that are performed on individual data items and tuples whereas in our case we assume data to be atomic and look at how it moves from one principal in a distributed system to another. These two different views are reflected in the formal frameworks used in each case. In databases, the formal frameworks used are either the Nested Relational Calculus [11], a core query language, or the $\lambda$-calculus [3], a core functional language. This means that the fundamental model of provenance in databases is that of functions mapping inputs to outputs. Provenance in this setting is simply a record of the relation between the inputs and outputs. This simple idea leads to multiple notions of provenance as we saw already, such as Where, Why and How provenance. In our work, we study distributed systems as formally modelled by an extension of the $\pi$-calculus [48, 49]. The basic building blocks of a distributed system are processes that run in parallel and communicate by message passing. It is this message passing that gives rise to our notion of provenance. As data is communicated from one principal in a distributed system to another, we wish to keep track of the principals that it has been at and what enabled them to transmit it.

The differences between the two settings are also reflected in the way provenance is represented. In databases, the representation of provenance takes several forms depending on the exact provenance notion, but generally ranges from simple labels, to sets of labels, to polynomials of labels. Each item in the input is given a unique label. Items in the output of a query are then labeled with a provenance annotation in order to record the items in the input that contributed to their presence in the output. In our case, we represent provenance as ordered, nested sequences. Each such sequence is composed of events which record the send and receive actions performed by principals in the distributed system. Each event records the principal that performed the action, the type of the action, and the provenance of the channel used.

Other than the provenance notion itself, our work considers the whole provenance management lifecycle and proposes a formal framework that integrates the definition of provenance, its tracking, querying as well as its security, all in one calculus. The Where, Why and How provenance notions and their formalisations only address two aspects of the provenance management lifecycle, that of defining provenance and that of tracking it. The approaches used for defining and tracking provenance, especially in the formal models of Cheney et al. [15, 16], are similar to ours in that they take the form of either instrumenting the operational semantics of the calculus to track provenance at run time or propose a static type system to approximate the provenance of data at compile time. The details of the operational semantics and the type system are of course different because we are dealing with different settings and different notions of provenance.

### 2.2.2 Workflows

#### 2.2.2.1 Provenance and reproducibility

Provenance has always been part of the scientific method, necessitated primarily by the exploratory nature of scientific research and the need for reproducibility. Scientists used logbooks to document the setup of their experiments and allow other researchers to more easily reproduce their results. Computers, like with many other tasks, brought automation to a large part of the scientific process. Not only that, but with the availability of vast computational resources, scientists are able to model more complicated processes and analyse larger sets of data. Documenting these with handwritten notes is an almost impossible task, and it is only natural that the same computational tools used to model processes and analyse data are used to capture their provenance.

#### 2.2.2.2 Scientific workflow systems

Although scientists can use (and have used) general purpose scripting languages and programming languages to model processes and analyse data, these lack several features that make them less than an ideal solution. In this regard, their main shortcomings lie perhaps in their high barrier to entry for non-programmers and their lack of built-in provenance management. Scientific workflow systems aim to address these shortcomings. They provide an environment where scientists can interactively create computational tasks, and where provenance information is tracked automatically by the underlying run time environment. In most scientific workflow systems, workflows are created by composing together building blocks known as *modules*. A module is an abstraction of an underlying service provided by a tool, a scientific library or a web service. As such, each module denotes a well defined unit of functionality such as reading input data from a file or a database, carrying out certain calculations on these data and producing graphs to visualise the results of calculations. A module has a number of *input ports* and a number of *output ports*. To create a workflow, multiple modules are composed together by *connecting* the output ports of one module to the input ports of another. These connections denote the flow of data from one module to another. Modules may be provided by the workflow system or by extension libraries, or may themselves be created as workflows from other modules, thus providing the ability to nest workflow definitions.

Scientific workflow systems provide an integrated environment in which scientists can create workflows, execute them, and capture their provenance all in a very seamless manner. The main role of these workflows is to provide an easy way for scientists to integrate disparate services, often provided by different groups and institutions. The scientist can add extra annotations to the static definition of the workflow, or to its run time executions. However, these only need to be done for aspects of the process which take place outside the workflow system; the provenance of the workflow itself is captured systematically by the workflow system.

#### 2.2.2.3 Types of workflow provenance

The definition of a workflow itself tells us what processes are involved and what the flow of data between them looks like, and as such can be considered a form of provenance. In the workflow provenance community, this type of provenance is known as *prospective provenance*, as it gives us, *a priori*, information about the steps that will be executed by a particular workflow. However, prospective provenance naturally lacks information that would only be available at run time. Examples of such information include the parameters of a particular execution of a workflow, the date and time when it was executed, and the data received from an external data source. *Retrospective provenance* fills this gap; it gives us information about a particular run of a workflow, including the run time aspects of the execution. It should be noted that the two types of provenance do not depend on each other, and hence access to or availability of one type of provenance does not imply access to or availability of the other.

Workflow systems can capture provenance at different levels of detail, and therefore like provenance in databases, it is possible to have different types of provenance depending on what granularity one chooses. Finer-grained provenance provides more details but may lead to large amounts of data and as a result would be harder to query and extract information from. The most natural level at which provenance can be captured is the workflow level itself. That is, capturing the different steps executed and the flow of data between them. As each step in the workflow may denote a library call, a remote service invocation or even the execution of a sub-workflow, it is also possible to capture provenance information about the internal state and execution of the step itself. This of course depends on the definition of the workflow step. If the step represents a library call then the code of this latter may be instrumented beforehand to capture its provenance at a logical level, or the Operating System may be relied on to capture low level system calls. In the case where the step represents the invocation of a remote service, then the workflow system would have to rely on this remote service to capture meaningful provenance and make it available with the results it provides. Finally, for sub-workflows, the workflow system can capture provenance the same way it does for the parent workflow.

#### 2.2.2.4 Example scientific workflow systems

There have been several projects with the aim of developing middleware to support provenance in different scientific domains. These include Chimera [22] (physics and astronomy), $^{my}$Grid [68] (biology), CMCS [60] (chemical sciences), and ESSW [23] (earth sciences). The motivation for supporting provenance in these domains is to allow scientists to understand, analyse and reproduce results of experiments. Of these systems we highlight $^{my}$Grid , which aims to provide service-based middleware for supporting *in silico* experiments in biology, with particular emphasis on the ability to share results of these experiments. Provenance is considered central for enabling this. $^{my}$Grid supports two types of provenance [30]. The first is called *derivation path* provenance, and records the process by which results were derived from inputs and would include such information as database queries, input parameters to programs and workflow descriptions. The second form of provenance consists of *annotations* attached to different kinds of entities, and may include standard annotations such as time of creation, date of last modification and owner, as well as custom annotations describing the entity with various concepts from the scientific domain.

Moreau et al. [53] advocate a vision where applications based on the Service Oriented Architecture (SOA) are *provenance-aware*, that is they allow provenance of data to be retrieved, analysed and reasoned over. To realise this vision, they introduce a *provenance life-cycle*, in which provenance-aware applications record *process documentation* and store it in a *provenance store*. The process documentation describes what happened at execution time while the provenance store offers a persistent and secure database, from which provenance information can be retrieved and analysed. The provenance store also offers capabilities for administering the provenance information. Process documentation may be composed of several *p-assertions*. These are assertions made by components of the process documenting their execution. Moreau et al. identify various kinds of p-assertions including: *interaction p-assertions*, documenting messages exchanged between the different services of the process; *relation p-assertions*, describing how the service obtained output data from input data; and *service state p-assertions*, which record the internal state of the service. Together, these different types of p-assertions can provide a complete documentation of the execution of a process.

Simmhan et al. [67], propose a taxonomy of data provenance characteristics and use it to compare several projects that investigate data provenance. They focus mainly on projects that deal with provenance for scientific workflows. Their taxonomy categorises provenance systems in terms of why they record provenance (provenance application), what entities they document (subject of provenance), how they represent provenance (provenance representation), how they store it (provenance storage) and what facilities they provide to access it (provenance dissemination).

#### 2.2.2.5 Comparison with our work

Works on provenance in workflow systems, as we saw in this section, have been largely practical. This contrasts with our approach which is mainly theoretical and aims to study the mathematical underpinnings of provenance in distributed systems. The provenance notions themselves are close to our own however, and at any rate, closer than those in databases. This similarity stems from the fact that workflow systems and Service Oriented Architectures can be considered as examples of distributed systems. Therefore, the provenance notions proposed in these settings are often coarse-grained, representing provenance in the form of graphs, and resembling to some extent our own representation of provenance.

To illustrate the similarity in how provenance is represented, consider for example the provenance-aware Service Oriented Architecture proposed by Moreau et al. [53]. The authors refer to their notion of provenance as process documentation. As we mentioned already, process documentation is composed of p-assertions. Moreau et al. identify three types of p-assertions:

- interaction p-assertions which record the messages that are exchanged between different services,
- relation p-assertions which record the relation between the messages that a service receives and the messages that it sends, and
- service state p-assertions which are meant to record the internal state of the service.

In our calculus, provenance is represented in the form of nested sequences. Each sequence is composed of events which are meant to record the send and receive actions of principals. Therefore, events are similar to interaction p-assertions in the provenance-aware Service Oriented Architecture. We do not have anything comparable to relation or service state p-assertions in our provenance notion however. The reason is that since the values in our model are atomic, the relation between what a process receives and what it sends is quite simple; it is the identity function since the value is never modified. We do not have any explicit notion of service state in our model. One could argue that the processes running at a principal and the values they have at any given point in time could be considered its state. We do not document this in provenance however since it is irrelevant to the applications of provenance we wish to study in this work.

### 2.2.3 Cross-cutting concerns

In this section we look at cross-cutting concerns, problems that are of interest to the wider provenance research community. We identify three such concerns, formalisation, security and standardisation. All of these are in fact concerns that reach far beyond provenance and are important in every field of Computer Science. However, they do need a special treatment in the context of provenance, requiring either a careful adaptation of the existing models and tools to provenance, or perhaps even a fresh look that takes into account any unique features and constraints that provenance tracking may give rise to. In the following we explain why.

#### 2.2.3.1 Formalisation

Formalisation is concerned with the development of formal models of provenance. The aim of such formal models is to bring mathematical rigour to the study and analysis of provenance tracking systems. Cheney et al. [15] were among the first to study provenance in a formal setting. Their approach draws on well understood and established topics from programming languages such as information flow analysis, dependency analysis and program slicing. They apply these programming language techniques to give a semantic definition of provenance. This semantic definition is based on a property they call *dependency-correctness*, which aims to capture the dependencies between parts of the output of a query and parts of its input. When annotations are said to be propagated in a dependency-correct manner by queries, it is meant that changes to parts of the input annotated with particular annotations will only result in changes to parts of the output annotated with those same annotations. Cheney et al. provide both a dynamic provenance tracking semantics as well as an approximation of it based on static type checking.

Acar et al. [1] develop a core calculus for provenance, based on a general purpose functional programming language. This promises to be more generic than previously proposed models that only target one field, such as workflows or databases. Their operational semantics captures traces, which they use to compare different provenance notions by providing suitable views over traces. They also explore two security properties known as obfuscation and disclosure which we cover in more detail in the next section.

#### 2.2.3.2 Security

The case for secure provenance is easy to make. On the one hand, if we are to become dependent on provenance information for making critical decisions, then we need to make sure that enough provenance information is available and has not been tampered with. On the other hand, since the provenance of a piece of data or an artifact reveals information about how that piece of data or artifact came to be, and who changed it or otherwise influenced it, then for privacy reasons we may not want that information leaked to unintended recipients. These are instances of problems that are the staple of security research in computer systems:

- Availability: information is available whenever it is needed.
- Integrity: information is not tampered with, or at least not undetectably.
- Confidentiality: information is not disclosed to unauthorised parties.

Research on provenance security aims to address these problems in the context of provenance. Provenance poses some interesting new challenges and that makes a straightforward adaptation of existing security models and tools not possible. The main challenge lies probably in the relationship between the artifact and its provenance. Provenance reveals information that describes the past states of the artifact, and in doing so, it might violate the security requirements of the artifact. This means that security models for provenance need to take into account both the security requirements of the provenance information as well as those of the underlying artifact.

Hasan et al. [32] and Braun et al. [7] were among the first to highlight the problems and challenges that provenance posed in terms of privacy and security. They argued that existing security models were not suitable to deal with these problems and challenges and advocated the need for research on provenance security. Similar problems were also raised by Davidson et al. [20, 21], where several key questions about privacy in the presence of provenance tracking and querying were framed.

Efforts to answer the aforementioned questions followed by those authors as well as others. Hasan et al. [33] consider the problem of ensuring the integrity of provenance against forgeries. They propose an architecture to accomplish this and demonstrate its viability by implementing a prototype library, Sprov, that tracks provenance of data reads and writes at the application level. They also carry out experiments to analyse the performance overhead of provenance tracking using their library and conclude that it is within acceptable limits for most applications.

Cheney [13] proposes a formal framework for provenance security. Within this high-level generic framework, the author gives definitions of several provenance properties, including provenance counterparts to the data availability and confidentiality properties, which the author terms *disclosure* and *obfuscation* respectively. Like availability and confidentiality, disclosure and obfuscation are meant to characterise what information principals are able to see and what information is withheld from them. Cheney also studies three different instances of the framework based on automata, databases, and workflows.

#### 2.2.3.3 Standardization

Aware of the plethora of provenance systems that have been developed, standardisation efforts aim to simplify interoperability between these systems by advocating common models of provenance. Of these we highlight the community led series of Provenance Challenges that resulted, among other things, in the proposal of the Open Provenance Model (OPM) [52]. The Open Provenance Model offers a technology-agnostic representation of provenance with the aim of enabling the exchange of provenance between different systems.

With the increase of interest in provenance on the Web, the World Wide Web Consortium (W3C) commissioned a study to understand the state-of-the-art in provenance. This culminated in the publication of the Provenance XG Final Report [26] and led to the formation of the Provenance Working Group. This latter is tasked with defining a language for exchanging provenance information among applications. As of the time of writing, the Provenance Working Group has been working on a family of specifications under the PROV umbrella [27]. This family of specifications includes a data model, PROV-DM [54]; a human-friendly notation for provenance, PROV-N [55]; an ontology, PROV-O [43]; a mechanism for accessing and querying provenance, PROV-AQ [40]; a formal semantics, PROV-SEM [14]; and an XML schema, PROV-XML [36].

#### 2.2.3.4 Comparison with our work

This section covered several works which we tried to group under the three headings of formalisation, security and standardisation. The works on formalisation and security are close to our own in terms of approach. We agree with them on the use of programming language semantics and static analysis to formalise provenance. The works of Cheney et al. [15] and Acar et al. [1] are based on the Nested Relational Calculus (NRC) and

Transparent ML (TML) respectively. These are suitable for studying provenance in systems that are based on a functional model of computing such as databases. However, they fall short when considering provenance in distributed systems and it is our opinion that a framework based on a process calculus such as the $\pi$-calculus is more suitable. The $\pi$-calculus allows us to model in a natural way fundamental features of distributed systems such as communication, parallel execution and non-determinism.

Standardisation efforts have focused mainly on practical issues. Their representation of provenance is quite close to ours however. Consider for example the Open Provenance Model (OPM) where provenance is represented as a directed graph. At this basic level, this is not very different from our model where provenance is represented as a nested, ordered sequence. OPM recognises three types of entities, artifacts, processes and agents. All three types of entities can be found in our model in some form. Plain values are our artifacts and they are the entities whose history is recorded by provenance. Processes and agents map to our notions of processes and principals respectively. OPM captures causal dependencies between the three types of entities in the form of different types of relationships. Examples of these includes:

- Process P *used* artifact A.
- Artifact A *was generated by* process P.
- Process P *was controlled by* agent A.

In the graph model of provenance, the three entities represent the nodes of the graph and the causal relationships its edges. Viewed as a graph, our provenance representation has two types of nodes; single principal names and provenance sequences. The principal names represent the authors of events while the provenance sequences represent the provenance of the channels used for sending and receiving data. Our provenance notion also has two types of edges; send edges to record send actions and receive edges to record receive actions.

The previous section also highlighted the PROV [27] family of specifications that is currently being developed by the W3C Provenance Working Group. This family of specifications aims to address the various aspects of provenance required to allow for the interoperable interchange of provenance information on the Web. At the heart of this family is the PROV-DM [54] data model. The PROV-DM specification aims to provide a common data model for the wide range of provenance applications on the Web. As such, it is quite generic, admitting notions of entities, activities, agents, roles, time and others. Our provenance sequences can be considered an instance of the PROV-DM data model. Our notions of values and principals can be represented as entities and agents in PROV-DM, while the ordering of events can be catered for by the notion of instantaneous time found in the PROV-DM specification. Concrete representations of the PROV-DM data model can be obtained in XML and RDF as defined by the PROV-XML [36] and PROV-O [43] specifications. In addition to these two, PROV-N [55] defines a more human friendly notation for provenance. Our provenance sequences can be encoded in many formats, including XML and RDF, however such encodings are orthogonal to the aims of the present work and therefore they are not pursued in this thesis. The PROV family of specifications also addresses the problem of interpreting provenance in PROV-DM by defining PROV-CONSTRAINTS [18], a set of constraints for PROV-DM, and PROV-SEM [14], a model-theoretic semantics for PROV-DM. Formal semantics is at the core of the present work. We define both operational semantics for the calculus as well as denotational semantics for provenance sequences. We also give formal definitions and proofs of all notions and properties used in this work. In addition to these specifications, the Provenance Working Group is also defining a specification for accessing and querying provenance, PROV-AQ [40]. Querying provenance in our calculus is achieved using pattern matching. We give an abstract definition of pattern matching languages and also define a concrete pattern matching language to use with examples. The sample pattern matching language we define is based on regular expression pattern matching [35], which can be used to query XML documents, and therefore, any data expressed in XML.

## 2.3 Concluding remarks

Comparing provenance in databases to that in workflows, it is possible to say that research in databases is concerned with fine-grained provenance while research in workflows is mainly interested in coarse-grained provenance. Provenance in databases considers operations on data at granularities ranging from complete databases and relations, to individual tuples and data items within those relations. In contrast, the provenance notions that have been studied by the workflow community focused on how datasets move between components of a single system and from one system to another. In this regard, workflow provenance is closer to our work; the provenance notion we study in this work assumes data is atomic and tracks its flow between different principals. In fact, the $\pi$-calculus, which forms the basis of our work, can be considered a formal model of workflows, and our work can be seen as a first attempt at formalising workflow provenance. We are careful not to overstate this though, as the aim of the present work is not to formalise existing models of provenance in workflow systems but rather to study provenance and related problems in the context of distributed systems.

Research on provenance in workflows has been largely practical, focusing on building systems and middleware to track and manage provenance information. Most theoretical work has come from the database community, spearheaded primarily by the works of Cheney et al. [1, 15, 16]. Our work is similar to these in spirit. We agree with those works in our use of programming language semantics and type systems to study provenance tracking. Their works are based on the lambda calculus or its variants however, while ours takes a process algebraic approach. In fact, to the best of our knowledge, our work is the first to study provenance using process calculi.

Several research works have looked at provenance security and we have already covered some of them in Section 2.2.3.2. In addition to the differences in setting and approach, our work also differs from those works in its aims. Our aim is to study the ramifications of provenance tracking on the privacy and security policies of principals and to develop programmatic tools that allow principals to control the disclosure of provenance.

# Chapter 3

# The Provenance Calculus

In this chapter, we present *the provenance calculus*, a formalism aimed at studying provenance in distributed systems. The provenance calculus is based on a variant of the asynchronous $\pi$-calculus extended with explicit *identities* with the aim of modelling multiple *trust domains*. This basic setting is then enriched with *provenance annotated data*, *dynamic provenance tracking* and *dynamic provenance checking*. This means that all data products (exchangeable entities) are seen as being essentially composed of two parts: the actual data content and a meta-data annotation representing the provenance of this content. Dynamic provenance tracking takes the form of a provenance tracking semantics which ensures that, as the system evolves, the provenance annotations of values are updated to reflect changes to the values or to their context. Finally, principals are able to specify their trust policies in the form of conditions on the provenance of values they are interested in consuming, and dynamic provenance checking enforces these policies at run time, ensuring principals only receive data with provenance that is compliant with these policies.

We elect to present the calculus in two steps. Firstly, in Section 3.1, we describe the plain version of the calculus, that is the version with no provenance tracking. This should allow us to focus on the workings of the calculus itself without the complications of provenance tracking. We then proceed, in Section 3.2, to present the provenance annotated version. In this latter version, all values are annotated with provenance information and the reduction relation is modified in order to update this information as the system evolves. Splitting the presentation this way makes explicit the two roles of the provenance tracking reduction relation, namely that of describing how the system evolves as its components interact, and that of tracking the provenance of the different values to reflect how the evolution of the system affects them. This means that the provenance tracking semantics should preserve the system transitions allowed by the plain semantics, or in other words, that the two semantics should coincide with respect to possible system transitions. As we will see, this would have indeed been true if the annotated version of the calculus *only* added provenance annotations and provenance tracking. However, in addition to this, our annotated version also adds provenance checking, allowing principals to decide what data to receive based on its provenance. This has the effect of ruling out system transitions that would result in principals receiving values with the wrong provenance, and hence means that the set of possible transitions in the annotated version is a subset of that in the plain version.

Section 3.3 defines a sample pattern language for use in provenance checking. It then gives several examples to illustrate the main features of the provenance calculus. The chapter concludes by discussing the design of the calculus and alternative choices of primitives.

## 3.1 Plain version

The plain version of the calculus is a variant of the asynchronous $\pi$-calculus [6, 34], a version of the $\pi$-calculus [48, 49] where message output is non-blocking, extended with explicit identities. As our aim is to use these identities for provenance tracking, we do not provide any other primitives for dealing with them, and hence they do not play any role in the plain version of the calculus. The calculus makes use of a form of input-guarded choice that is restricted to a single channel (similar in spirit to the one used by Castagna et al. [12]). This results in simpler semantics than the full version of input-guarded choice; nevertheless it still allows for both internal choice and external choice in the annotated version of the calculus.

The plain version of the calculus is aimed to represent the computational core of our calculus, allowing us to model the essence of computation in distributed systems. For example, the following term:

$$a[m\,\langle v\rangle] \mid b[m\,(x).P]$$

denotes two principals or agents, running in parallel. These two principals do not need to be co-located, and any interaction between them takes place via message passing. The principal $a[m\,\langle v\rangle]$ has the name $a$ and is running code $m\,\langle v\rangle$. We call such code a process. Executing this process will send the value $v$ to be received by any principal that may happen to be listening on channel $m$. In our calculus, as we will see later, we denote such a value that is ready to be consumed by the term $m\langle\!\langle v\rangle\!\rangle$. A principal listening on $m$, such as $b[m\,(x).P]$, would receive this value once it has been made available. When this principal receives the value $v$, the result is denoted by the term $b[P\{^v/_x\}]$, indicating that all free occurrences of the placeholder variable $x$ in process $P$ have been substituted by the value $v$ just received. Execution of the resulting process will proceed according to similar rules.

The above simple example illustrates the main computational features of the calculus. For expressivity, we also include guarded choice for non-determinism, matching for conditional branching, replication for repetitive behaviour, and restriction of channel names for lexical scoping. The formal syntax and semantics of the calculus is given in the following sections.

### 3.1.1 Syntax

We assume a set $\mathcal{X}$ of *variables*, ranged over by $x, y, z, \ldots$, a set $C$ of *channel names*, ranged over by $l, m, n, \ldots$, and a set $\mathcal{A}$ of *principal names*, ranged over by $a, b, c, \ldots$. We assume that all three sets are pair-wise disjoint and define the set $\mathcal{V}$ of *values* to be $C \cup \mathcal{A}$, and use the letters $u, v, \ldots$ to range over this set. We also define the set $\mathcal{I}$ of identifiers as $\mathcal{V} \cup \mathcal{X}$ and use the meta-variables $w, w'$ to range over identifiers.

#### 3.1.1.1 Processes

Processes are ranged over by $P, Q, \ldots$ and their formal syntax is summarised in Figure 3.1. The primitive $m\,\langle v\rangle$ denotes a process that is ready to output the value $v$ on channel $m$. We use a variant of input-guarded choice that is restricted to a single input channel. This simplifies the provenance tracking reduction relation (given in Section 3.2) and enables us to focus on the use of provenance. The input-guarded choice or summation, written as $\Sigma_{i\in I} m\,(x).P_i$ for some finite indexing set $I$, denotes a process that may receive some value $v$ on channel $m$ and continue as $P_j$ after substituting the value $v$ just received for the formal parameter $x$. The continuation process $P_j$, where $j \in I$, is chosen non-deterministically. As customary, we use 0 as syntactic sugar for $\Sigma_{i\in\emptyset} m\,(x).P_i$ and $m\,(x).P_1 + m\,(x).P_2$ as syntactic sugar for $\Sigma_{i\in\{1,2\}} m\,(x).P_i$. Matching, $\mathsf{if}\ v = v'\ \mathsf{then}\ P\ \mathsf{else}\ Q$ , denotes the process that proceeds as $P$ if $v$ is equal to $v'$ and as $Q$ otherwise. Scope restriction of channel $n$ to process $P$ is denoted by $(\nu n)P$ while parallel composition of processes $P$ and $Q$ is denoted by $P \mid Q$. The process $*\,P$ behaves as an infinite number of copies of $P$ running in parallel.

Figure 3.1. *Plain syntax: processes*

| | |
|---|---|
| $P, Q ::=$ | process terms |
| $w\,\langle w' \rangle$ | output of value $w'$ on channel $w$ |
| $\Sigma_{i \in I} w\,(x).P_i$ | input-guarded choice |
| $\mathsf{if}\ w = w'\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | equality test of $w$ and $w'$ |
| $(\nu n)P$ | private channel $n$ with scope $P$ |
| $P \mid Q$ | parallel composition of $P$ and $Q$ |
| $* P$ | replication of $P$ |

#### 3.1.1.2 Systems

A *system* is the composition of zero or more *located processes* and *messages*. We use $S, T, \ldots$ to range over systems and summarise the syntax of systems in Figure 3.2. The simplest system is 0 which is used to denote the empty system. A located process, $a[P]$, stands for a process $P$ that is running under the authority of a principal $a$. A message is a value that has been sent but not yet received, and is denoted in the calculus by $n\langle\!\langle w \rangle\!\rangle$. Restriction is denoted by $(\nu n)S$ while parallel composition of two systems is denoted by $S \mid T$.

Figure 3.2. *Plain syntax: systems*

| | |
|---|---|
| $S, T ::=$ | system terms |
| $a[P]$ | located process |
| $n\langle\!\langle w \rangle\!\rangle$ | message |
| $(\nu n)S$ | channel $n$ with scope $S$ |
| $S \mid T$ | parallel composition of $S$ and $T$ |
| $0$ | empty system |

#### 3.1.1.3 Binders, substitution and $\alpha$-conversion

The variable $x$ in $\Sigma_{i \in I} w\,(x).P_i$ binds occurrences of $x$ in the continuations $P_i$ while the name $n$ in $(\nu n)P$ binds occurrences of $n$ in $P$. All other occurrences of variables and names are considered free. We use $fv(P)$, $bv(P)$, $fn(P)$ and $bn(P)$ for the sets of free variables, bound variables, free names and bound names in process $P$ respectively. We use $\equiv_{p\alpha}$ for the $\alpha$-equivalence relation, that is $P \equiv_{p\alpha} Q$ means that $P$ and $Q$ may be derived from each other by the change of bound variables and names. A substitution is a finite mapping from variables to identifiers. We use $\sigma, \sigma'$ to range over substitutions and write $\{^{w_1,\ldots,w_n}/_{x_1,\ldots,x_n}\}$ for the substitution of $w_i$ for $x_i$ for all $i \in 1 \ldots n$. We denote by $P\sigma$ the process obtained by applying the capture avoiding substitution $\sigma$ to process $P$. We call a process with one or more free variables an *open* process and a process with no free variables a *closed* process. All these notions extend to systems in a straightforward way.

### 3.1.2 Reduction semantics

The semantics of the calculus is defined by two relations, the *plain structural congruence relation* $\equiv_p$, and the *plain reduction relation* $\rightarrow_p$. Structural congruence allows us to make structural manipulations of systems which makes the definition of reduction simpler. The definition of structural congruence depends on the definition of *contexts* and *congruences*, which we give below.

**Definition 3.1** (Context)**.** The set of system contexts $C$ (contexts for short) is given by the following syntax:

$$C \quad ::= \quad [.] \quad | \quad (\nu n)C \quad | \quad C \,|\, S \quad | \quad S \,|\, C$$

The term $C[S]$ denotes the system obtained by replacing the occurrence of the hole [.] with the system $S$. Note that the grammar of contexts ensures that there is exactly one hole in every context.

**Definition 3.2** (Congruence)**.** An equivalence relation $R$ on systems is contextual, or a congruence, if whenever $(S, T) \in R$ then $(C[S], C[T]) \in R$ for all contexts $C$.

**Definition 3.3** (Structural congruence)**.** Structural congruence $\equiv_p$ is the smallest congruence on systems such that the axioms of Figure 3.3 hold.

Figure 3.3. *Plain semantics: structural congruence*

| | |
|---|---|
| (PStr Alpha) | $S \equiv_{p\alpha} T \Rightarrow S \equiv_p T$ |
| (PStr Par Nil) | $S \mid 0 \equiv_p S$ |
| (PStr Par Comm) | $S \mid T \equiv_p T \mid S$ |
| (PStr Par Assoc) | $(R \mid S) \mid T \equiv_p R \mid (S \mid T)$ |
| (PStr Res Nil) | $(\nu n)0 \equiv_p 0$ |

| | |
|---|---|
| (PStr Res Res) | $(\nu n)(\nu m)S \equiv_p (\nu m)(\nu n)S$ |
| (PStr Res Par) | $((\nu n)S) \mid T \equiv_p (\nu n)(S \mid T) \quad \text{if } n \notin fn(T)$ |
| (PStr Sys Nil) | $a[0] \equiv_p 0$ |
| (PStr Sys Par) | $a[P \mid Q] \equiv_p a[P] \mid a[Q]$ |
| (PStr Sys Res) | $a[(\nu n)P] \equiv_p (\nu n)(a[P])$ |
| (PStr Sys Rep) | $a[* P] \equiv_p a[P \mid * P]$ |

The rule PStr Alpha states that two systems are structurally congruent if they are $\alpha$-convertible. The three rules PStr Par Nil, PStr Par Comm, and PStr Par Assoc state that $(S, |)$ is a commutative monoid with identity element 0. The rules PStr Res Nil, PStr Res Res, and PStr Res Par describe properties of restriction. They state that redundant restrictions may be garbage collected, that the order of restrictions is immaterial and that the scope of a restriction may be *extruded* to include parallel systems. The last four rules describe properties of located processes. The rule PStr Sys Nil states the equivalence of $a[0]$ and 0, and hence like PStr Res Nil with redundant restrictions, it may be seen as stating that redundant principal identifiers may be garbage collected. The rule PStr Sys Par says that the located process $a[P \mid Q]$ is the same as the parallel composition of the two located processes $a[P] \mid a[Q]$, or in other words that principal identifiers distribute over parallel composition. The rule PStr Sys Res says that the scope of names may be moved from the process level to the system level or vice versa. The rule PStr Sys Rep states that replication of a process denotes an infinite number of copies of that process running in parallel.

Having defined structural congruence, we now describe how systems may evolve by defining the reduction relation.

**Definition 3.4.** The reduction relation $\to_p$ is defined by the rules of Figure 3.4.

Figure 3.4. *Plain semantics: reduction*

PRed Snd

$$\frac{}{a[m\langle v\rangle] \to_p m\langle\!\langle v\rangle\!\rangle}$$

PRed Rcv

$$\frac{j \in I}{a[\Sigma_{i\in I} m(x).P_i] \mid m\langle\!\langle v\rangle\!\rangle \to_p a[P_j\{^v/_x\}]}$$

PRed If$_t$

$$\frac{}{a[\text{if } m = m \text{ then } P \text{ else } Q\ ] \to_p a[P]}$$

PRed If$_f$

$$\frac{m \text{ is not equal to } n}{a[\text{if } m = n \text{ then } P \text{ else } Q\ ] \to_p a[Q]}$$

PRed Res $\dfrac{S \rightarrow_p S'}{(\nu n)S \rightarrow_p (\nu n)S'}$ PRed Par $\dfrac{S \rightarrow_p S'}{S \mid T \rightarrow_p S' \mid T}$ PRed Str $\dfrac{S \equiv_p T \quad T \rightarrow_p T' \quad T' \equiv_p S'}{S \rightarrow_p S'}$

The two main reduction rules are PRed Snd and PRed Rcv, which describe the sending and receiving of values respectively. Note that communication is split into these two steps in the plain version of the calculus to match the semantics of the annotated version. There, this is done as it simplifies the semantics and more closely matches our intuitions on the nature of communication and provenance tracking in distributed systems; a value is first packaged and addressed to its intended recipients, and then it is received and consumed. In the rule PRed Snd, a located process $a[m\,\langle v\rangle]$ may output the value $v$ on the channel $m$ resulting in the message $m\langle\langle v\rangle\rangle$. In the rule PRed Rcv, a message $m\langle\langle v\rangle\rangle$ may be received by the located process $a[\Sigma_{i\in I} m\,(x).P_i]$, which then continues as $a[P_j\{^v/_x\}]$. The process $P_j$, where $j \in I$, is chosen non-deterministically. The rules PRed IF$_t$ and PRed IF$_f$ give the semantics of name matching, they state that the process if $u = v$ then $P$ else $Q$ continues as $P$ if the two values being tested are equal (PRed IF$_t$) and as $Q$ otherwise (PRed IF$_f$). The other three rules are standard and state that reduction is preserved under restriction, system composition as well as under structural congruence.

## 3.2 Annotated version

Having defined the computational features of the calculus in the form of the plain version, we are now ready to describe the annotated version, taking this computational core and extending it with provenance management. To integrate provenance management into the calculus, the annotated version adds provenance annotations to all values, instruments the semantics of the calculus to update these annotations as the computation proceeds, and enables principals to use the provenance annotations to decide which data to receive and how to use it.

The simple example of the previous section, when expressed in the annotated version of the calculus, would look something like this:

$$a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \mid b[m{:}\kappa'_m\,(\{a,b,c\}!\mathsf{Any}\ ;\ \mathsf{Any}\ \mathsf{as}\ x).P]$$

The first difference to notice about this system is that the two occurrences of the channel name $m$ and the one occurrence of the value $p$ are now annotated or tagged with the provenance sequences $\kappa_m$, $\kappa'_m$ and $\kappa_p$. These are meant to represent their histories up to that state in the evolution of the system. Executing the process at $a$ would give us the message $m\langle\!\langle p : a!\kappa_m \,;\, \kappa_p\rangle\!\rangle$. This packaged message is different from that seen in the previous section, and the difference lies in the presence of provenance and in the fact that this provenance changed from $\kappa_p$ to $a!\kappa_m \,;\, \kappa_p$, indicating that the value $p$ was sent by $a$ on a channel whose provenance was $\kappa_m$, and that before this, the value $p$ had provenance $\kappa_p$. This is how the provenance of values is kept up-to-date with the evolution of the system. Another difference to note is that the input construct now takes the form $m{:}\kappa'_m(\{a, b, c\}!\mathsf{Any} \,;\, \mathsf{Any} \text{ }\mathsf{as}\text{ } x)$. This denotes that principal $b$ is listening on channel $m$ just like before. However, this time, it is putting a condition on the values that it would like to receive on channel $m$, namely, that their provenance must satisfy the pattern $\{a, b, c\}!\mathsf{Any} \,;\, \mathsf{Any}$. This pattern is satisfied by the provenance of any value that came from either $a$, $b$ or $c$. Since the value $p$ would be sent by $a$ in this example, its provenance would satisfy the condition set by $b$ and hence it would be received by $b$. The result of $b$ consuming the value is $b[P\{^{p:b?\kappa'_m;a!\kappa_m;\kappa_p}/_{x}\}]$. Notice how the provenance of the value $p$ changed again to reflect that it was received by principal $b$ on a channel with provenance $\kappa'_m$. Provenance annotated values, the tracking of provenance at run time, and the ability to specify conditions on the provenance of values to be received are the three main features the annotated version of the calculus adds on top of the foundation laid down in the previous section.

## 3.2.1 Syntax

Syntactically, the annotated version differs from the plain one in two aspects: (1) tagging values with provenance information, and (2) adding a second parameter, a pattern, to the input construct. We describe these in the following.

### 3.2.1.1 Annotated values

All data values are now annotated with meta-data representing their provenance. This is reflected in the introduction of *annotated values*, terms of the form $p : \kappa$. An annotated value $p : \kappa$ denotes a plain value $p$ with provenance annotation $\kappa$. Plain values are the values of the plain calculus and are ranged over by $p, q, \ldots$ in the annotated calculus. We reserve the letters $v, u, \ldots$ for annotated values.

We represent the provenance of a value as a sequence of events. The events in a provenance sequence are assumed to be temporally ordered from left to right, with the left-most event (the head of the sequence) being the most recent. We use $\mathcal{K}$ for the set of provenance sequences and $\mathcal{E}$ for the set of events and let $\kappa, \kappa', \ldots$ and $e, e', \ldots$ range over elements of each set respectively. We denote the empty provenance sequence by $\epsilon$, the singleton sequence composed of event $e$ only by $e$ and the concatenation of sequences $\kappa$ and $\kappa'$ by $\kappa; \kappa'$. The concatenation operator ; is associative, so $(\kappa ; \kappa') ; \kappa''$ and $\kappa ; (\kappa' ; \kappa'')$ denote the same provenance sequence. It is not commutative however as our sequences are ordered. The empty provenance sequence $\epsilon$ is the unit of ; so the sequences $\kappa ; \epsilon$ and $\epsilon ; \kappa$ are both equivalent to the sequence $\kappa$. We have two types of events: output events, written as $a!\kappa$, and input events, written as $a?\kappa$. An output event $a!\kappa$ in the provenance of a value denotes that the value has been sent by principal $a$ on a channel whose provenance is $\kappa$, while an input event $a?\kappa$ denotes that the value has been received by principal $a$ on a channel whose provenance is $\kappa$.

Note how the structure of provenance sequences ensures that the requirements of provenance described in Section 1.1.4 are indeed satisfied. In particular, the principal that originally produced the value would be recorded as the author of the right-most event, satisfying the **Originator** requirement of provenance. **Transmitters** of the value are all the principals that appear as authors of events at the top level in the provenance of the value. Then, nested within each event, we find the provenance of all the channels used for communicating this value. These record any principal that supplied a channel and therefore acted as an **Enabler** in the communication of the value. To meet the **Type** requirement, provenance sequences contain two types of events, send events that record send actions and receive events that record receive actions. Finally, as already mentioned, provenance sequences are ordered from right to left and therefore they satisfy the **Order** requirement. To illustrate these points, consider the example from the previous section, and in particular the annotated value $p : b?\kappa'_m ; a!\kappa_m ; \kappa_p$. To make this concrete, assume that the provenance sequence $\kappa'_m$ is equivalent to $b?\epsilon ; d!\epsilon ; \epsilon$, the provenance sequence $\kappa_m$ is equivalent to $a?\epsilon ; c!\epsilon ; \epsilon$ and that the provenance sequence $\kappa_p$ is equivalent to $\epsilon$. Therefore, the value $p$ has the following provenance $b?(b?\epsilon ; d!\epsilon ; \epsilon) ; a!(a?\epsilon ; c!\epsilon ; \epsilon) ; \epsilon$. The right-most event in this provenance sequence is $a!(a?\epsilon ; c!\epsilon ; \epsilon)$ and therefore the originator of the value is principal $a$. The value in this case was transmitted from its originator, principal $a$, directly to the principal where it is currently, principal $b$. Hence, principals $a$ and $b$ are the only two transmitters of the value so far. The channels used by $b$ and $a$ for communicating the value were obtained from $d$ and $c$ respectively as indicated by their provenance sequences $b?\epsilon ; d!\epsilon ; \epsilon$ and $a?\epsilon ; c!\epsilon ; \epsilon$. Principals $d$ and $c$ therefore enabled $b$ and $a$ to communicate by providing them with copies of the communication channel $m$. The events in this provenance sequence fall in one of two types, either send events such as $c!\epsilon$, or receive events such as $a?\epsilon$. The events at each level in the provenance sequence are ordered from right to left.

#### 3.2.1.2 Patterns

The second syntactic change lies in modifying the input-guarded choice construct to take pattern specifications $\pi$. Patterns $\pi$ are used by principals to specify the provenance of data they are willing to receive on a particular channel and to branch to different continuations in response to different provenance sequences. The modified choice construct is written as $\Sigma_{i \in I} w\,(\pi_i \text{ as } x).P_i$. Instead of defining a particular *pattern matching language*, we opt for a more general approach and make the calculus parametric on the choice of the pattern matching language. We do give a concrete language to use with the examples however. We give the definition of pattern matching languages in Definition 3.5 and summarise the syntactic changes in Figure 3.5.

**Definition 3.5.** A pattern matching language is a pair $(\Pi, \models)$ where $\Pi$ is a set of patterns, ranged over by $\pi, \pi', \ldots$, and $\models\, \subseteq \mathcal{K} \times \Pi$ is the pattern satisfaction (or matching) relation, a relation between provenance sequences and patterns.

Figure 3.5. *Annotated syntax: summary of changes*

| | |
|---|---|
| $P, Q ::=$ | process terms |
| $w\,\langle w' \rangle$ | output of value $w'$ on channel $w$ |
| $\Sigma_{i \in I} w\,(\pi_i \text{ as } x).P_i$ | input-guarded choice with patterns |
| $\text{if } w = w' \text{ then } P \text{ else } Q$ | equality test of $w$ and $w'$ |
| $(\nu n)P$ | private channel $n$ with scope $P$ |
| $P \mid Q$ | parallel composition of $P$ and $Q$ |
| $* P$ | replication of $P$ |
| $\kappa ::=$ | provenance sequences |
| $\epsilon$ | empty provenance |
| $e$ | single event |
| $\kappa\,;\,\kappa$ | sequential composition |
| $e ::=$ | events |
| $a!\kappa$ | output event |
| $a?\kappa$ | input event |

| | |
|---|---|
| $p, q, \ldots \in \mathcal{W} \triangleq \mathcal{C} \cup \mathcal{A}$ | plain values |
| $u, v, \ldots \in \mathcal{V} \triangleq p : \kappa$ | annotated values |
| $w, w', \ldots \in \mathcal{I} \triangleq \mathcal{V} \cup \mathcal{X}$ | identifiers |
| $\pi, \pi', \ldots \in \Pi$ | patterns |

### 3.2.2 Provenance tracking semantics

The definition of structural congruence for annotated systems, denoted by $\equiv_a$, is a straightforward adaptation of that for plain systems and hence it is omitted. It is worth noting however that names in scope restriction, such as $n$ in $(\nu n)S$, appear in plain form as restriction is an operator of the calculus. Within one such scope, a channel name may have multiple occurrences, each with a possibly different provenance sequence reflecting the history of that particular copy. An annotated channel name $n : \kappa$ is considered bound if it occurs within the scope of a restriction $(\nu n)$.

The reduction relation in the annotated version of the calculus, which we denote by $\rightarrow_a$ and often refer to as the *provenance tracking* reduction relation, extends that of the plain calculus with two main features. The first is that it updates the provenance of the relevant values after each reduction step and the second is that it allows principals to use patterns to restrict the set of values they are willing to receive on a particular channel. Hence, the provenance tracking reduction relation can be seen as performing two tasks. On the one hand, it prescribes the computational semantics of the calculus. On the other hand, it keeps track of provenance and ensures provenance policies are complied with. We define the provenance tracking reduction relation in Figure 3.6.

Figure 3.6. *Annotated semantics: provenance tracking reduction*

ARED SND

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p{:}a!\kappa_m \,;\, \kappa_p\rangle\!\rangle}$$

ARED RCV

$$\frac{j \in I \qquad \kappa_p \models \pi_j}{a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \;\mathsf{as}\; x).P_i] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \rightarrow_a a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

ARED IF$_t$

$$\frac{}{a[\mathsf{if}\; m{:}\kappa = m{:}\kappa' \;\mathsf{then}\; P \;\mathsf{else}\; Q\;] \rightarrow_a a[P]}$$

ARED IF$_f$

$$\frac{m \text{ is not equal to } n}{a[\mathsf{if}\; m{:}\kappa = n{:}\kappa' \;\mathsf{then}\; P \;\mathsf{else}\; Q\;] \rightarrow_a a[Q]}$$

ARed Res
$$\frac{S \rightarrow_a S'}{(\nu n)S \rightarrow_a (\nu n)S'}$$

ARed Par
$$\frac{S \rightarrow_a S'}{S \mid T \rightarrow_a S' \mid T}$$

ARed Str
$$\frac{S \equiv_a T \quad T \rightarrow_a T' \quad T' \equiv_a S'}{S \rightarrow_a S'}$$

In the rule ARed Snd, a located process $a[m{:}\kappa_m \langle p{:}\kappa_p \rangle]$ may output the value $p : \kappa_p$ on the channel $m : \kappa_m$ which results in the message $m\langle\langle p : a!\kappa_m \,;\, \kappa_p \rangle\rangle$. What should be noted here is that, in addition to describing the sending of the value $p$ as in the plain version, the rule also describes how the provenance of $p$ changes from $\kappa_p$ to $a!\kappa_m \,;\, \kappa_p$ after the output action to reflect the fact that the value has been most recently sent by principal $a$ on a channel whose provenance is $\kappa_m$. Note also that, since we are interested in using provenance as a means for deriving trust in the quality of data, and since we see trust fundamentally as a relation defined between principals, we only keep track of the principals involved in the action and ignore the channel name used. In the rule ARed Rcv, a message $m\langle\langle p{:}\kappa_p \rangle\rangle$ may be received by the located process $a[\Sigma_{i \in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i]$ if the provenance of the value satisfies one of the patterns $\pi_i$. The process $P_j$ whose pattern $\pi_j$ is satisfied is chosen for the continuation. If more than one such pattern exists, one of them is chosen non-deterministically. Note that here too, the provenance of the value $p$ is updated from $\kappa_p$ to $a?\kappa_m \,;\, \kappa_p$ to reflect the fact that it has most recently been received by principal $a$ on a channel with provenance $\kappa_m$. The rules ARed IF$_t$ and ARed IF$_f$ give the semantics of matching. It should be noted here that only the plain values are tested for equality while their provenance is ignored.[1] This means that if the two plain values are equal, irrespective of their provenances, the process in the then branch is chosen for the continuation as indicated by the rule ARed IF$_t$. If the two plain values are not equal, then the process in the else branch is chosen for the continuation as indicated by the rule ARed IF$_f$. Save for the addition of the provenance annotations and the provenance test in the input rule, the rest of the calculus and its semantics are the same as in the plain version. The other three rules are the standard ones for restriction, parallel composition and structural congruence.

## 3.3 Examples

In this section, we will give a few examples to illustrate the use of the provenance calculus. As the calculus is parametric on the choice of the pattern language, we start by

[1]Depending on the intended application, the provenance of the two values tested could be useful and hence would be tracked in the continuation. This is not considered here however as it is not important for the aims of the present work.

defining a sample pattern matching language first, which we then use in the subsequent sections to code our examples.

### 3.3.1 A sample pattern matching language

As our provenance sequences have a tree structure similar to that of XML documents, we choose to base our sample pattern language on regular expression pattern matching [35]. It is worth noting however that, since we do not allow variable bindings in our patterns at the moment, our patterns more closely resemble regular expression types and our pattern matching is more akin to dynamic type checking. The formal syntax of the language is given in Figure 3.7 where $X$ ranges over a countably infinite set of pattern names.

Figure 3.7. *Sample pattern matching language: syntax*

| | | | |
|---|---|---|---|
| $\pi ::=$ | patterns | $\alpha ::=$ | event patterns |
| $X$ | pattern name | $G!\pi$ | output |
| $\epsilon$ | empty sequence | $G?\pi$ | input |
| $\alpha$ | single event | $G ::=$ | group expressions |
| $\pi \,;\, \pi$ | sequence | $a$ | single principal |
| $\pi \vee \pi$ | alternation | $\sim$ | all principals |
| $\pi*$ | repetition | $G + G$ | group union |
| | | $G - G$ | group difference |

Pattern names allow us to express recursive patterns. They are interpreted with respect to a global set of definitions $\mathcal{D}_\Pi$ which associates with every pattern name $X$ a definition of the following form:

$$\mathsf{pat}\ X = \pi$$

The set $\mathcal{D}_\Pi$ is seen as a mapping from pattern names to their definitions. We use $dom(\mathcal{D}_\Pi)$ for the set of pattern names defined by $\mathcal{D}_\Pi$ and $\mathcal{D}_\Pi(X)$ for the body of the definition of pattern name $X$. The definition of a pattern name may include other pattern names in its body which allows for the specification of recursive patterns. To guarantee that patterns correspond to regular tree automata, we require a well-formedness condition that disallows recursion at the top level. Formally, we require that:

$$\forall X \in dom(\mathcal{D}_{\Pi}).X \notin reach(\mathcal{D}_\Pi(X))$$

where $reach(\pi)$ is defined to be the smallest set satisfying:

$$reach(X) = \{X\} \cup reach(\mathcal{D}_\Pi(X)) \qquad reach(\epsilon) = \emptyset \qquad reach(\alpha) = \emptyset$$
$$reach(\pi*) = reach(\pi) \qquad reach(\pi \,;\, \pi') = reach(\pi) \cup reach(\pi')$$
$$reach(\pi \vee \pi') = reach(\pi) \cup reach(\pi')$$

The formal semantics of patterns is given by the satisfaction relation $\models\, : \Pi \times \mathcal{K}$ and the group denotation function $[\![-]\!] : \mathcal{G} \to \mathcal{A}$. These are defined in Figure 3.8.

Figure 3.8. *Sample pattern matching language: semantics*

Definition of the satisfaction relation ($\models$).

Sat Name
$$\frac{\mathcal{D}_\Pi(X) = \pi \quad \kappa \models \pi}{\kappa \models X}$$

Sat Emp
$$\frac{}{\epsilon \models \epsilon}$$

Sat Snd
$$\frac{a \in [\![G]\!] \quad \kappa \models \pi}{a!\kappa \models G!\pi}$$

Sat Rcv
$$\frac{a \in [\![G]\!] \quad \kappa \models \pi}{a?\kappa \models G?\pi}$$

Sat AltL
$$\frac{\kappa \models \pi}{\kappa \models \pi \vee \pi'}$$

Sat AltR
$$\frac{\kappa \models \pi'}{\kappa \models \pi \vee \pi'}$$

Sat Cat
$$\frac{\kappa \models \pi \quad \kappa' \models \pi'}{\kappa \,;\, \kappa' \models \pi \,;\, \pi'}$$

Sat Rep
$$\frac{\forall i \in 1 \ldots n . \kappa_i \models \pi}{\kappa_1 \,;\, \ldots \,;\, \kappa_n \models \pi*}$$

Definition of the denotation function ($[\![-]\!]$).

$$[\![a]\!] = \{a\} \qquad [\![\sim]\!] = \mathcal{A} \qquad [\![G + G']\!] = [\![G]\!] \cup [\![G']\!] \qquad [\![G - G']\!] = [\![G]\!] \setminus [\![G']\!]$$

A pattern name $X$ matches any provenance sequence that is matched by the body of its definition. The pattern $\epsilon$ matches the empty provenance sequence (denoted by $\epsilon$ as well). The two patterns $G!\pi$ and $G?\pi$ match send and receive events respectively. The use of *group expressions* $G$ in these patterns allows us to perform more general tests against the principal that performed the event. The group expression $a$ denotes the singleton set containing principal $a$ only, while $\sim$ denotes the set of all principals. $G + G'$ and $G - G'$ denote union and difference of groups respectively. The pattern $\pi \,;\, \pi'$ matches a provenance sequence that is composed of two parts that match $\pi$ and $\pi'$ respectively. The alternation of patterns $\pi$ and $\pi'$, denoted by $\pi \vee \pi'$, matches a sequence that matches either patterns, while the repetition of pattern $\pi$, denoted by $\pi*$, matches any provenance sequence that is the composition of zero or more sub-sequences, each of which matches the pattern $\pi$.

To simplify the examples, we define the regular expression forms $\pi?$ and $\pi+$ as syntactic sugar for $\pi \vee \epsilon$ and $\pi ; \pi*$ respectively. We also define the following pattern names.

$$\mathsf{AnyEvent} = \sim!\mathsf{Any} \vee \sim?\mathsf{Any}$$
$$\mathsf{Any} = \mathsf{AnyEvent}*$$

which match any event and any provenance sequence respectively. Note that the two patterns are defined by mutual recursion and are well-defined since $\mathsf{AnyEvent}^*$ denotes either $\epsilon$ or one or more occurrences of $\mathsf{AnyEvent}$. To ease readability, we also elide the empty provenance sequence $\epsilon$ from annotated data and the pattern $\mathsf{Any}$ from input constructs.

### 3.3.2 Example systems

#### 3.3.2.1 Authentication

Probably the simplest use of provenance is to establish the authenticity and integrity of messages. For example, in the following system, principal $a$ checks that the data it is receiving on channel $m$ is coming from principal $b$ intact.

$$a[m\,(\mathsf{Any} ; b!\mathsf{Any} ; \mathsf{Any}\ \mathsf{as}\ x).P] \mid S$$

Note how this is accomplished using the pattern $\mathsf{Any} ; b!\mathsf{Any} ; \mathsf{Any}$, which is satisfied by any value that went through principal $b$ at some point in its route to $a$. This is the case because values in our calculus are atomic. The only operations that principals may perform on them are either to use them as communication channels or send them to other principals. Therefore, a value with provenance $\mathsf{Any};b!\mathsf{Any};\mathsf{Any}$ is a value that originated somewhere and was transmitted by principal $b$ (possibly among other principals) before it was received by the current principal.

#### 3.3.2.2 Trust policies and data quality

More elaborate uses of provenance are also possible. For example, principals may express their trust policies as patterns representing what sources they consider trustworthy and hence produce high quality data. Consider, for instance, the following system:

$$a[m\,(\mathsf{Any} ; b!\mathsf{Any}\ \mathsf{as}\ x).P \mid n\,(\mathsf{Any} ; c!\mathsf{Any}\ \mathsf{as}\ y).Q]$$

In this system, principal $a$ only accepts data that originated at $b$ on channel $m$. This is accomplished using the pattern $\mathsf{Any}\ ;\ b!\mathsf{Any}$ which is matched by any provenance sequence that ends with an event of the form $b!\kappa$ denoting that the value originated at principal $b$. On the other hand, principal $a$ only accepts data originating at $c$ on channel $n$. This may represent the local trust policy of principal $a$, reflecting the fact that $a$ considers $b$ to produce high quality data for consumption on $m$ and similarly for $c$ and channel $n$. The channels here may represent different kinds of data; for example channel $m$ may be a used for communicating scholarly papers while channel $n$ may be used by $a$ for consuming entertainment. In this case, the above patterns denote that $a$ considers $b$ to be a good source of scholarly work while $c$ is reserved for entertainment only.

As another example, consider the following system:

$$b[l\,(a!\mathsf{Any}\ \mathsf{as}\ x).P + l\,((\sim - a)!\mathsf{Any}\ \mathsf{as}\ y).Q]$$

where provenance is used to determine what to do with data received on channel $l$. If the data is coming directly from principal $a$, and it has not been through any other principal, then it is processed by $P$, otherwise it is processed by $Q$. This could be because $a$ represents a source whose credibility or accuracy is highly regarded, and hence data authored by this principal is allowed to be used for the highly critical process $P$. On the other hand, other sources are not considered as such and hence their data is only suitable for the less critical process denoted by $Q$. Naturally, a different principal might have a different policy, considering principal $a$, a source of high quality data from $b$'s point of view, as untrustworthy. In this case, this principal would employ different patterns to determine what data to consume, possibly reversing those used by $b$ if its trust beliefs were completely the opposite of those of $b$.

Many other provenance policies can be expressed. For example, one can consider policies that discriminate not only based on the top level provenance of the value, but also goes deeper and discriminates based on the provenance of the channels used.

### 3.3.2.3 Auditing

Provenance can also be used as an auditing and troubleshooting tool to establish who might have been responsible for an error. For example, in the following system:

$$S \triangleq a[m\,\langle v\rangle] \mid s[m\,(x).n'\,\langle x\rangle] \mid c[n'\,(x).P] \mid b[n''\,(x).Q]$$

principal $a$ is trying to send a value $v$ to principal $b$, and this has to be done through an intermediary $s$ (because $a$ does not have a direct link to $b$ for example). Because of faulty code at $s$, the value gets forwarded to $c$ instead, as indicated by the following reduction:

$$S \Rightarrow_a c[P\{^{v:c?\epsilon;s!\epsilon;s?\epsilon;a!\epsilon}/_{x}\}] \mid b[n''\,(x).Q]$$

Now when $c$ detects the error, perhaps due to the unexpected value, $c$ can use the provenance $c?\epsilon; s!\epsilon; s?\epsilon; a!\epsilon$ to tell what principals were involved in making this error. In this case, $a$, $s$, and $c$ itself. The three principals may be further investigated to determine who and what exactly caused the error.

#### 3.3.2.4 Photography competition

The following example describes a photography competition. Contestants submit their entries for the competition to the organiser of the competition, who forwards those entries to the appropriate judges. Each judge rates the entries allocated to them and returns the results to the organiser. The organiser then publishes the results and announces the winners. We consider a version of the competition with three contestants: $c_1$, $c_2$ and $c_3$, one organiser: $o$, and two judges: $j_1$ and $j_2$. The contestants submit their entries to the organiser on channel *sub* and receive the published results on channel *pub*. The organiser forwards entries submitted by $c_1$ and $c_3$ to judge $j_1$ and the entry by $c_2$ to $j_2$. The judges return the entries together with their ratings to the organiser. Note that below we are using polyadic versions of the send and receive constructs; such an extension to the calculus should be straightforward.

$$\mathbf{C}(c, \mathit{entry}, P) \triangleq c[\mathit{sub}\,\langle \mathit{entry}\rangle \mid \mathit{pub}\,(\mathsf{Any}\ ;\ c!\mathsf{Any}\ \mathsf{as}\ x\ ,\ \ \mathsf{Any}\ \mathsf{as}\ y).P]$$

$$\mathbf{O} \triangleq o[*\,(\Sigma_{i\in\{1,2\}}\mathit{sub}\,(\pi_i\ \mathsf{as}\ x).\mathit{in}_i\,\langle x\rangle \mid \mathit{res}\,(y,z).*\,\mathit{pub}\,\langle y,z\rangle)]$$

$$\mathbf{J}(j, \mathit{inc}) \triangleq j[\mathit{inc}\,(x).\mathit{res}\,\langle x, \mathit{rate}(x)\rangle]$$

$$\mathbf{Comp} \triangleq \mathbf{C}(c_1, e_1, P_1) \mid \mathbf{C}(c_2, e_2, P_2) \mid \mathbf{C}(c_3, e_3, P_3) \mid \mathbf{O} \mid \mathbf{J}(j_1, \mathit{in}_1) \mid \mathbf{J}(j_2, \mathit{in}_2)$$

where $\pi_1 \triangleq (c_1 + c_3)!\mathsf{Any}\ ;\ \mathsf{Any}$ and $\pi_2 \triangleq c_2!\mathsf{Any}\ ;\ \mathsf{Any}$. The system above evolves as follows, where $\Pi_{i\in\{1,\ldots,n\}}S_i$ is used to denote $S_1 \mid \ldots \mid S_n$.

$$\mathbf{Comp} \rightarrow_a^{*} \Pi_{i\in\{1,2,3\}}o[*\,\mathit{pub}\,\langle e_i : \kappa_{ei}, \mathit{rate}(e_i) : \kappa_{ri}\rangle] \mid \mathbf{O} \mid \Pi_{i\in\{1,2,3\}}c_i[P_i\{^{e_i:\kappa'_{ei}, \mathit{rate}(e_i):\kappa'_{ri}}/_{x,y}\}]$$

where:

$$\kappa_{ei} \triangleq \begin{cases} o?\epsilon \,;\, j_1!\epsilon \,;\, j_1?\epsilon \,;\, o!\epsilon \,;\, o?\epsilon \,;\, c_i!\epsilon & \text{if } i \in \{1,3\} \\ o?\epsilon \,;\, j_2!\epsilon \,;\, j_2?\epsilon \,;\, o!\epsilon \,;\, o?\epsilon \,;\, c_i!\epsilon & \text{if } i \in \{2\} \end{cases}$$

$$\kappa_{ri} \triangleq \begin{cases} o?\epsilon \,;\, j_1!\epsilon & \text{if } i \in \{1,3\} \\ o?\epsilon \,;\, j_2!\epsilon & \text{if } i \in \{2\} \end{cases}$$

$$\kappa'_{ei} \triangleq c_i?\epsilon \,;\, o!\epsilon \,;\, \kappa_{ei}$$

$$\kappa'_{ri} \triangleq c_i?\epsilon \,;\, o!\epsilon \,;\, \kappa_{ri}$$

After receiving the results from the judges, the organiser publishes them on channel *pub* as shown in the replicated output processes $* pub\,\langle e_i : \kappa_{ei}, rate(e_i) : \kappa_{ri}\rangle$ for $i \in \{1, 2, 3\}$. Every contestant listens on channel *pub* to receive their own results. This is achieved by the input construct $pub\,(\mathsf{Any}\,;\, c!\mathsf{Any}\ \mathsf{as}\ x\ ,\ \mathsf{Any}\ \mathsf{as}\ y)$. The pattern specifications allow the contestant to receive the pair containing the result for their own entry.

## 3.4 Concluding remarks

### 3.4.1 Input-guarded choice

The restriction of input-guarded choice to a single input channel makes the calculus simpler (and probably leads to an easier implementation). It is also closer to our intuitions and is expressive enough for our purposes. In fact, it still gives us both external and internal choice in the annotated version of the calculus. External choice manifests itself when the environment provides values with different provenances while internal choice is exhibited when a provenance sequence satisfies more than one pattern in a particular choice construct.

### 3.4.2 Patterns in the input construct

Principals make use of provenance by specifying their policies in the form of provenance patterns in the input construct. Together with the choice primitive, this allows them to use provenance to decide what data to consume and how to use it. This means that for a given channel, there are likely to be multiple incompatible patterns specified by different principals for what data each principal deems suitable for their own consumption. This is not a problem of course as the semantics makes sure, through dynamic provenance tests, that each principal only receives data that matches their requirements. Intuitively, this corresponds to a model of "unregulated markets"; that is if we think of each channel as modelling a market for exchanging data of some kind or for some purpose, then we see that there are no restrictions on the provenance of data each channel may carry. This is evident from the syntax of the restriction construct, $(\nu n)P$, where a channel $n$ is created with no conditions on the provenance of data that may be exchanged on it. It is because of this lack of "regulation" that each principal has to check the provenance of the data before they consume it. One can envisage a different model, which we refer to as that of "regulated markets", where each channel, besides any possible restrictions on the kind of data, also has restrictions on the provenance of the data it may carry. This would require each channel to be mapped to a single pattern specifying the provenance of data that may be communicated on it. Hence, the syntax of restriction would be modified to $(\nu n : \pi)P$ (as would be expected for a standard type system) and the "implementation" would be either in terms of a static provenance tracking system that guarantees that only data with provenance that complies with the policy of a channel is exchanged along that channel, or in terms of a run time check that only allows output operations to proceed if they would result in data with the correct provenance. In the latter model, transitions would take place between configurations of the form $\Gamma \vdash S$ where $S$ is a system and $\Gamma$ an environment recording the provenance policies (i.e., the types) of each free channel name in $S$. For example, the send and receive rules may be written as follows:

ARed Snd

$$\frac{a!\kappa_m\,;\kappa_p \models \Gamma(m)}{\Gamma \vdash a[m{:}\kappa_m\,\langle p : \kappa_p\rangle] \rightarrow_a \Gamma \vdash m\langle\!\langle p : a!\kappa_m\,;\kappa_p\rangle\!\rangle}$$

ARed Rcv

$$\frac{j \in I}{\Gamma \vdash a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_a \Gamma \vdash a[P_j\{^{p:a?k_m;\kappa_p}/_x\}]}$$

As the rules show, output is only allowed if the data has provenance that conforms to the provenance policy of the channel. This allows principals to safely receive data without making any checks of their own.

### 3.4.3 Querying provenance

The choice of input-guarded sum as the primitive for querying provenance was influenced mainly by our view of patterns as acting like policies that specify what data sources each principal deems trustworthy. As we mentioned earlier, we think of each channel as representing a market for exchanging data of a particular kind. Each principal has their own policies on what combinations of sources produce high quality data. Having these as patterns in the input construct means that principals only receive data that matches their provenance policies. One could instead opt for a different model where principals receive any data they find available on a particular channel, after which they can then use a separate primitive for checking the provenance of the data received and determine what to do with it. For example, a primitive that would allow branching based on the shape of the provenance sequence of a value could be written as follows:

$$\textsf{case}\ w\ \textsf{of}\ \pi\ \textsf{then}\ P\ \textsf{else}\ Q$$

and its semantics may be given by the following two rules:

$$\textsc{ARed Case}_t \quad \frac{\kappa \models \pi}{\textsf{case}\ p : \kappa\ \textsf{of}\ \pi\ \textsf{then}\ P\ \textsf{else}\ Q \ \rightarrow_a P} \qquad \textsc{ARed Case}_f \quad \frac{\kappa \not\models \pi}{\textsf{case}\ p : \kappa\ \textsf{of}\ \pi\ \textsf{then}\ P\ \textsf{else}\ Q \ \rightarrow_a Q}$$

A principal could still choose to send the data back on the channel it received it from if it does not match its provenance policy. This would, however, "pollute" the provenance of the value with provenance information referring to this principal. This is one of the reasons why patterns in the input construct were deemed to offer a better primitive. Note that, depending on the facilities offered by the pattern matching language, our version of the input construct would often be strictly more expressive than the branching primitive demonstrated above, in which case this latter may be encoded using our input-guarded choice construct.

### 3.4.4 Implementation concerns

The calculus presented in this chapter should be viewed as an abstract model of provenance management systems. It does not address implementation concerns such as space and time efficiency. In fact, a naive implementation of the calculus would provide for a very inefficient system. There are two main reasons for this. The first is that provenance annotations keep expanding without bound as new events are added to them. We expect a concrete implementation of the calculus to put a limit on how large provenance annotations can get. This limit can be expressed either in terms of the size of the provenance sequence (i.e. as a provenance sequence reaches a certain size, older events are archived or deleted) or in terms of its age (i.e. all events older than a certain age are archived or deleted irrespective of the size of the provenance sequence). The second reason is that annotated data is *passed by value* in the calculus. That is, a copy of the annotated value is made when communication happens. This may be inefficient for large provenance sequences, especially in cases where multiple values share common ancestry. To see what we mean here, consider the following simple system:

$$a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \mid b[m{:}\kappa'_m\,(x).(n{:}\kappa_n\,\langle x\rangle \mid l{:}\kappa_l\,\langle x\rangle)]$$

This system is composed of two principals, a principal $a$ that sends a value $p$ on channel $m$ and a principal $b$ that is ready to receive the value and make two copies of it, one it intends to send on channel $n$ and the other on channel $l$. After two transitions, we should get the following:

$$b[n{:}\kappa_n\,\langle p : b?\kappa'_m\,;a!\kappa_m\,;\kappa_p\rangle \mid l{:}\kappa_l\,\langle p : b?\kappa'_m\,;a!\kappa_m\,;\kappa_p\rangle]$$

where we have two copies of the annotated value $p : b?\kappa'_m\,;a!\kappa_m\,;\kappa_p$ at principal $b$. Naively, this would require double the memory footprint. However, there is no reason why an implementation could not instead store only one copy of the provenance sequence $b?\kappa'_m\,;a!\kappa_m\,;\kappa_p$ and simply have pointers to it from principal $b$. This could also be used when communicating a value from one principal to another where only a pointer to the provenance sequence is passed instead of the actual provenance sequence. These are just some of the examples of optimizations that an implementation might perform. The calculus itself is only an abstract model and therefore it does not address issues relating to the efficient storage and communication of provenance.

# Chapter 4

# The Role of Provenance

The aim of this chapter is to study the role of provenance in our calculus. It does that by comparing and contrasting the two versions of the calculus, the plain version and the annotated version. These two, differing only in the provenance management features added to the annotated version, make it possible for us to quite easily highlight the role that provenance plays in the calculus.

The ability of principals to access provenance in the annotated version depends on the patterns they employ. These in turn depend on the specific pattern matching language we decide to use and its expressive power. Therefore, we start the chapter with Section 4.1, where we look at pattern matching languages and study their properties. After highlighting the different classes of pattern languages and their impact on principals' ability to discern different aspects of provenance, we move on to compare the two versions of the calculus. We first draw attention to the syntactic differences between the two versions, namely, the annotation of all values with their provenance and the addition of patterns to the input construct. Naturally, semantic differences are more important than syntactic ones. This is especially the case since all annotated systems can be translated to plain ones by simply *erasing* the provenance annotations and the input patterns. We use two approaches to analyse the semantic differences between the two versions:

- Comparing the system transitions allowed by the reduction relation of each of the two versions of the calculus. This is carried out in Section 4.2.
- Comparing the discriminating power of the canonical behavioural equivalence of each of the two versions of the calculus. This is done in Section 4.3.

We find that plain systems have more transitions than annotated ones in general. This is because the annotated reduction relation only allows input transitions when the provenance of values satisfies one of the patterns specified by the principal. This means that, in terms of system transitions, the behaviour of annotated systems can be seen as a subset of that of plain ones. In a way, it is the safe subset; the subset of behaviours where principals only receive data perceived by them as being of acceptably high quality given its provenance. Behavioural equivalences give us another angle from which to compare the two versions. We find that the equivalence obtained in the case of the plain version is more coarse, admitting more systems as being equivalent, than the one obtained in the annotated version. This is because the provenance of values published by a particular system reveals information about its internal structure; information that would not have been visible otherwise to an outside observer. This is consistent with the results of the first approach since it is the prohibition of certain system transitions that gives the equivalence more discriminating power in the annotated version. The remainder of this chapter offers more details about this.

## 4.1 Pattern languages

The definition of pattern languages we gave in Definition 3.5 is quite generic and imposes hardly any conditions at all on what constitutes a pattern language. The sample pattern language defined in Section 3.3.1 is fairly expressive; it allows principals to discriminate between values based on the presence or absence of specific patterns in their provenance. It is actually very expressive, as expressive as one would want any pattern language to be in fact as we will demonstrate later in this section. The patterns it offers range from the very stringent, accepting only *one* provenance sequence, to the very broad, accepting *all* provenance sequences. It also allows the specification of *unsatisfiable* patterns; ones with no matching provenance sequences and which when used would mean the input action of the principal would never be exercised. We do not anticipate many uses for such patterns, nevertheless they are allowed by the grammar of the sample pattern language.

### 4.1.1 Separating the tracking of provenance from its usage

What is interesting about the genericity of the pattern language definition is that it allows one to envisage a wide range of pattern languages, each useful for a particular class of applications. This is important since the pattern language is really the *interface* between the implementation of provenance tracking and the usage of provenance by principals.

Through the specification of a pattern language, it is possible to provide an abstract view of provenance and hide from principals as much of the implementation details of provenance tracking as is desired. Not only that, but different notions of provenance can be achieved in one of two ways:

- either by varying the amount of provenance tracked by the provenance tracking reduction relation.
- or by varying the amount of provenance made available to principals through the specification of a suitable pattern language.

To see how this would work, consider our annotated calculus from Section 3.2. There, provenance sequences record the send and receive actions of principals. For each such action, they record the principal name, the type of the action, and the provenance of the channel used. This was what our provenance requirements called for. Were our requirements different, say we were interested instead in a provenance notion where only the author of the action (i.e. the name of the principal executing the action) was needed, then we could have been able to get away with recording far less provenance. This would correspond to changing the definition of the provenance reduction relation so that it only recorded the author of an action. Specifically, the send and receive rules would need to be modified as follows:

$$\textsc{ARed Snd Alt}\ \frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p : a\,;\,\kappa_p\rangle\!\rangle}$$

$$\textsc{ARed Rcv Alt}\ \frac{j \in I \qquad \kappa_m \models \pi_j}{a[\Sigma_{i\in I} m\,(\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \rightarrow_a a[P_j\{^{p:\kappa_p}/_x\}]}$$

As can be seen from the above two rules, only the send action records provenance now, and not only that, but it only records the principal name. The reason is that we only need to know the list of principals that a value has travelled through to reach its current location, its route so to speak, and as such recording the name of the principal after the receive action would be redundant. The structure of provenance becomes much simpler too, a sequence of principal names of the form $a_1\,;\ldots;a_n$. Naturally, the pattern language we would use for such a case would take that into account.

Although the above approach might look like the natural way to implement a provenance notion, it is possible to achieve the same effect in a more abstract way, using a suitable pattern language as we already pointed out. This would involve keeping the same definition of provenance tracking given in Section 3.2 and instead using a pattern matching language that only exposed the route of the value to principals. What is attractive about this approach is that we can vary the pattern language to obtain different provenance notions without having to modify our definition of provenance tracking, thus keeping the calculus of Chapter 3 and any results relating to it intact. For example, to expose only the route of the value, we can use patterns of the form $G_1$ ; ... ; $G_2$, where $G_i$ denotes a group expression similar to those defined for our sample pattern language. The semantics of these patterns would be given with respect to our original definition of nested provenance sequences, as follows:

$$\textsc{Sat Snd Alt}\,\frac{a \in [\![G]\!]}{a!\kappa \models G} \qquad \textsc{Sat Rcv Alt}\,\frac{a \in [\![G]\!]}{a?\kappa \models G}$$

We only need to modify the definition of satisfaction for send and receive events. The other rules should remain unchanged. As the rules show, although provenance events still record the author's name, the event type, and the provenance of the channel used, the pattern language only looks at the name of the author and ignores the rest. This way, principals see provenance as though it was simply a sequence of principal names, much the same as in the above approach. Hence, the two approaches are equivalent in terms of the provenance they expose to principals, and so allow exactly the same set of system transitions.

### 4.1.2 Properties of pattern matching languages

We already commented informally on the expressive power of our sample pattern language. Here, we give a brief formal review of some of its properties, drawing attention to some alternative pattern languages.

**Definition 4.1.** Let $\mathcal{M} \triangleq (\Pi_{\mathcal{M}}, \models_{\mathcal{M}})$ be a pattern matching language respecting Definition 3.5 and let $\pi$ be a pattern in $\Pi_{\mathcal{M}}$. We say that:

- Pattern $\pi$ is satisfiable if there exists a provenance sequence $\kappa$ such that $\kappa \models_{\mathcal{M}} \pi$, pattern $\pi$ is unsatisfiable if this does not hold.
- Pattern $\pi$ is valid if for all provenance sequences $\kappa$, it holds that $\kappa \models_{\mathcal{M}} \pi$, pattern $\pi$ is invalid if this does not hold.
- Pattern $\pi$ uniquely identifies a provenance sequence $\kappa$ if $\kappa \models_{\mathcal{M}} \pi$ and for all provenance sequences $\kappa'$ it holds that if $\kappa' \models_{\mathcal{M}} \pi$ then $\kappa = \kappa'$.
- Language $\mathcal{M}$ admits satisfiable (respectively unsatisfiable, valid, invalid) provenance sequences if there exists a provenance sequence $\kappa$ such that $\kappa$ is satisfiable (respectively unsatisfiable, valid, invalid).

- Language $\mathcal{M}$ is expressively complete if every provenance sequence $\kappa$ can be uniquely identified by a pattern $\pi$ in $\Pi_{\mathcal{M}}$.

These properties are meant to give us a measure of how expressive a particular pattern language is. The first four (satisfiability, validity and their negations) are elementary properties; we expect all useful patterns in any nontrivial pattern matching language to be satisfiable as the use of unsatisfiable patterns would simply block input actions. Additionally, we expect almost all of them to be invalid as otherwise we wouldn't be able to discriminate between different provenance sequences. A pattern that uniquely identifies a particular provenance sequence can distinguish this sequence from every other provenance sequence. A language that is expressively complete gives principals the ability to discriminate between any two provenance sequences, no matter how small the differences between them are. Therefore, it allows principals to express very precise provenance policies. Conversely, for a pattern language to be useful, it also needs to include patterns with less discriminating power. That way, principals are also able to express less stringent provenance policies. Hence, an ideal pattern matching language, in our view, is one that offers principals a good balance between the ability to express strict policies on the one hand, and more lenient ones on the other, allowing for the expression of a wide range of provenance policies.

A language with only one pattern and where the pattern is either unsatisfiable or valid is called a *provenance-unaware* pattern matching language. Note that all unsatisfiable patterns can be considered semantically equivalent and hence having more than one would be redundant. This is also the case for multiple valid patterns. We use $\mathcal{M}_0$ to denote the provenance-unaware language with the unsatisfiable pattern and $\mathcal{M}_1$ to denote the provenance-unaware language with the valid pattern. Although these two languages are too inexpressive for most provenance policies of interest, they are useful tools for highlighting some properties of provenance. We end this section by formally proving some of the properties of our sample pattern matching language.

**Proposition 4.2.** *The sample pattern matching language of Section 3.3.1 admits satisfiable, unsatisfiable, valid and invalid patterns, and it is expressively complete.*

*Proof.* The first part of the proposition is easy to prove as we simply need to give an example of each type of pattern. For satisfiability, the pattern $a!\mathsf{Any}$ is satisfied by the provenance sequence $a!\epsilon$. It is invalid however as it does not match the provenance sequence $a?\mathsf{Any}$. Both of these can be easily demonstrated using the satisfaction rules given in Figure 3.8. For an example of an unsatisfiable pattern, consider $(a-a)!\epsilon$, which equates to saying that we need a value that was sent by $a$ but at the same time not sent by $a$, which clearly can not be satisfied by any provenance sequence. An example of a valid pattern is Any. To prove unsatisfiability and validity, we have to show that the pattern $(a - a)!\epsilon$ matches no provenance sequences, and that the pattern Any matches every provenance sequence. We can do that by induction on the structure of provenance sequences. The details are straightforward and hence omitted for brevity.

To show that our sample pattern language is expressively complete, we need to show that every provenance sequence has a unique pattern that matches it. It turns out this is very easy in the case of our sample pattern language as patterns are a superset of provenance sequences, and hence we can use the provenance sequence itself as a pattern that uniquely identifies it. □

We conclude this section by noting that the properties of patterns we have discussed are defined relative to the *syntax* of provenance sequences. This is important to point out as not all syntactically legal provenance sequences are actually semantically meaningful. The provenance sequence $a!\epsilon\ ;\ a!\epsilon\ ;\ \epsilon$ for example is allowed by the grammar of provenance sequences, but it is semantically incorrect as it would never arise from a transition in our calculus. The reason for this is that send and receive actions always interleave in our calculus contrary to what this provenance sequence says. This impacts both satisfiability and invalidity and means that some patterns deemed satisfiable or invalid according to the above definitions may actually be unsatisfiable or valid if only semantically correct provenance sequences are considered. To illustrate this with an example, consider the pattern $a!\mathsf{Any}\ ;\ a!\mathsf{Any}\ ;\ \mathsf{Any}$ which is satisfied by the aforementioned provenance sequence. This pattern is however unsatisfiable since the provenance sequence $a!\epsilon;a!\epsilon;\epsilon$ would never arise in a legal transition in our calculus, nor would any other provenance sequence that satisfies the pattern. Chapter 5 discusses the semantics of provenance sequences and the reader is referred to that chapter for more details on this.

## 4.2 Relating the two versions of the calculus

The plain version of our calculus, presented in Section 3.1, is meant to give the computational core of the calculus. This computational core was extended with provenance management features in the annotated version in Section 3.2. The aim of making this separation explicit is to allow us to more easily highlight the role of provenance management and its impact on the calculus. The extensions introduced by the annotated version of the calculus consist of three features: (1) annotated values, (2) provenance tracking, and (3) provenance queries or policies. As we have seen, the latter take the form of patterns specified by each principal restricting what data they may consume on a particular channel, and given the flexibility of the input-guarded sum construct, may also be used to branch to different continuations based on the provenance sequence of the value received. Intuitively, this corresponds to deciding how to use the data based on how trustworthy it is deemed to be. More fundamentally however, the annotated version may be seen as simply ruling out those transitions (input transitions to be precise) which may lead to principals receiving data from sources they consider untrustworthy. That is, the provenance tracking reduction relation is "faithful" to the computational semantics of the calculus as given in the plain version, and only alters it by updating the provenance of values after each "safe" transition. Transitions which may lead to principals receiving data that violates their provenance policies are deemed unsafe and hence excluded through dynamic provenance checking. This is the semantics we expect to find in most instances of our calculus. However, as the annotated version is parametric on the choice of a pattern language, the exact behaviour of input actions will depend on the pattern language used.

To formalise the relation between the two versions, we define a set of *annotation erasure functions* which map annotated terms to their plain counterparts. More specifically, we define three such functions: one for identifiers, one for processes and one for systems. With a slight abuse of notation, we use $|{-}|$ to denote all three functions. We also define a "complementary" function on annotated values which returns the provenance annotation of a given annotated value. We call this latter the *annotation extraction function* and denote it by $\|{-}\|$. The definitions are given below.

**Definition 4.3** (Annotation erasure function)**.** The annotation erasure functions for identifiers, processes and systems are defined by induction as shown in Figure 4.1.

Figure 4.1. *Annotation erasure function*

IDENTIFIERS.

$|p : \kappa| \triangleq p$

$|x| \triangleq x$

SYSTEMS.

$|a[P]| \triangleq a[|P|]$

$|n\langle\!\langle w \rangle\!\rangle| \triangleq n\langle\!\langle |w| \rangle\!\rangle$

$|(\nu n)S| \triangleq (\nu n)|S|$

$|S \mid T| \triangleq |S| \mid |T|$

$|0| \triangleq 0$

PROCESSES.

$|w\,\langle w' \rangle| \triangleq |w|\,\langle |w'| \rangle$

$|\Sigma_{i\in I} w\,(\pi_i \text{ as } x).P_i| \triangleq \Sigma_{i\in I} |w|\,(x).|P_i|$

$|\text{if } w = w' \text{ then } P \text{ else } Q\,| \triangleq \text{if } |w| = |w'| \text{ then } |P| \text{ else } |Q|$

$|(\nu n)P| \triangleq (\nu n)|P|$

$|P \mid Q| \triangleq |P| \mid |Q|$

$|{*}\,P| \triangleq {*}\,|P|$

**Definition 4.4** (Annotation extraction function)**.** The annotation extraction function $\|{-}\| : \mathcal{V} \to \mathcal{K}$, which given an annotated value, returns its provenance annotation, is defined by the following clause: $\|p : \kappa\| \triangleq \kappa$.

Save for provenance tracking and checking, the two reduction systems of the plain and annotated versions are very similar. Provenance tracking itself does not actually impact the computational behaviour of the calculus; on the other hand, the checking of provenance against the policies of principals does have an impact. That impact depends on the pattern matching language we choose. However, no matter what pattern language is chosen, we know that the annotated version does not introduce any new behaviour. This is formalised in Theorem 4.5, which states that all system transitions derivable using the provenance tracking reduction relation $\to_a$ may also be derived using the plain reduction relation $\to_p$. As we have already commented, the inverse is not true in general since the provenance tracking semantics only allows input transitions if the data to be received has provenance that complies with the recipient's policies. This means that if we were to weaken the provenance policies of principals to the point where every provenance sequence is compliant, then we would expect the transitions of the annotated version to perfectly match those of the plain version. We have already seen a pattern matching language that achieves exactly this and that is $\mathcal{M}_1$. Recall that $\mathcal{M}_1$ is the pattern language with the singleton set of patterns, and whose single pattern is valid, matching every provenance sequence. Formalising this, Theorem 4.7 states that, in the instance of the calculus with pattern language $\mathcal{M}_1$, the provenance tracking reduction relation is completely faithful to the semantics of the plain version. We call such a reduction relation a *pure* provenance tracking reduction relation as it only tracks provenance but does not let principals check it to determine what data to receive nor otherwise use it. Note that even in such a setting, provenance tracking would still be useful as it may be used at the *meta* level for a wide range of applications such as auditing, troubleshooting and studying the properties of systems.

**Theorem 4.5** (Soundness of the provenance tracking semantics)**.** *$S \to_a T$ implies $|S| \to_p |T|$.*

*Proof.* Let us assume that $S \to_a T$ holds. We proceed by induction on the last rule used in its derivation. In each case, a corresponding rule exists in the plain version which we can use to derive $|S| \to_p |T|$. The rules are exactly the same in the two relations except for ARed Rcv. This latter has the additional premise $\kappa_p \models \pi_j$ in the annotated version. A corresponding derivation may be made in the plain version by simply dropping this premise. □

**Definition 4.6** (Pure provenance tracking)**.** Let $\rightarrow_{ppt}$ denote the instance of the reduction relation $\rightarrow_a$ where the pattern matching language used is $\mathcal{M}_1$.

**Theorem 4.7** (Completeness of the pure provenance tracking semantics)**.** *$S \rightarrow_{ppt} T$ implies $|S| \rightarrow_p |T|$. Vice versa, $|S| \rightarrow_p T$ implies $S \rightarrow_{ppt} T'$ for some $T'$ such that $|T'| = T$.*

*Proof.* The first part, $S \rightarrow_{ppt} T$ implies $|S| \rightarrow_p |T|$, is just an instance of the more general statement proved in Theorem 4.5 and so its truth follows from that theorem. The proof for the other direction, that is $|S| \rightarrow_p T$ implies $S \rightarrow_{ppt} T'$ for some $T'$ such that $|T'| = T$, also follows along similar lines to that of Theorem 4.5. For each case, we pick an annotated system $S$ and consider its erasure $|S|$ and the corresponding plain transition $|S| \rightarrow_p T$. We then apply the counterpart of the this transition in the annotated version to obtain $S \rightarrow_{ppt} T'$. This is possible since for each transition (except the input one) in the annotated version , its counterpart in the plain version has similar premises and a similar conclusion. It is also possible for the input rule as the annotated version has one extra premise, $\kappa_p \models_{\mathcal{M}} \pi_j$, but as we know, all $\pi_j$ patterns in $\mathcal{M}_1$ are actually the same and are equivalent to the valid pattern, and so that premise will always be true. This means that we can derive the input transition too. Deriving the annotated transition, we would find that the erasure of the system obtained this way $|T'|$ is indeed equal to the system obtained in the plain version $T$. □

## 4.3 Behavioural equivalences

The previous section looked at the difference between the two versions of the calculus by directly comparing their reduction relations. That highlighted the role of provenance in the calculus, namely that of ruling out transitions deemed unsafe as they would lead if allowed to principals consuming low quality data. In this section, we compare and contrast the two versions of the calculus again, but this time using behavioural equivalences. The aim is to use the same framework as the basis for developing two behavioural equivalences, one for each version of the calculus, thus obtaining easily comparable equivalences. By contrasting the discriminating power of these two equivalences, we should be able to highlight the role of provenance since any differences in discriminating power would be attributable to the addition of provenance in the annotated version.

Generally, behavioural equivalences provide a deep semantic understanding of a process calculus by defining what systems are considered the same in terms of their behaviour. Most often, we are only interested in the *observable* behaviour of systems, abstracting away from any *internal* behaviour. The reason for this should be clear; internal behaviour is invisible from the outside and any differences between two systems in their internal behaviour would not be discernible by interacting with them. In other words, the arbitrator of equivalence is itself seen as a system subject to the same rules of the calculus, and hence it is restricted in comparing systems to interacting with them according to the reduction rules of the calculus.

To give an example, consider the two plain systems $S$ and $T$ defined as follows:

$$S \triangleq a[n\langle p\rangle] \qquad\qquad T \triangleq (\nu m)(b[m\langle p\rangle] \mid c[m(x).n\langle x\rangle])$$

System $S$ consists of a single principal $a$ whose behaviour is simply to send the value $p$ on channel $n$. On the other hand, $T$ is relatively more complex, it consists of two principals $b$ and $c$; principal $b$ sends the value $p$ on a *private* channel $m$ to principal $c$ who then publishes it on channel $n$. In the plain version of the calculus, this is demonstrated by the transitions of the two systems given below:

$$\begin{aligned} S &\rightarrow_p n\langle\!\langle p\rangle\!\rangle & \qquad T &\rightarrow_p (\nu m)(m\langle\!\langle p\rangle\!\rangle \mid c[m(x).n\langle x\rangle]) \\ & & &\rightarrow_p c[n\langle p\rangle] \\ & & &\rightarrow_p n\langle\!\langle p\rangle\!\rangle \end{aligned}$$

Now consider another system $O$ acting as an arbitrator or observer. The job of system $O$ is to employ all the means it has at its disposal to try and tell systems $S$ and $T$ apart. This means that its behaviour can be arbitrarily complex as long as it can be expressed as a system in our calculus. We find that the only interaction $O$ can have with $S$ and $T$ is to receive the value $p$ on channel $n$. The observer $O$ cannot interact with either $S$ or $T$ in any other way. It is impossible for it to interact with $T$ on channel $m$ as it is a private channel. It is also impossible for it to detect the names of principals it is interacting with, nor can it detect the behaviour of principal $b$ in system $T$ as $b$ only exhibits internal or private behaviour. Moreover, although $T$ takes three steps to publish $p$ on $n$ compared to the single step taken by $S$, it is again impossible for $O$ to detect this “delay” as the calculus provides no means to do that. Therefore, $O$ will have but to proclaim $S$ and $T$ equivalent.

The discriminating power of observers changes when we add provenance however. To demonstrate that, consider the same two systems $S$ and $T$ but now expressed in the annotated version of the calculus. We assume all values have empty provenance to begin with and that principal $c$ in $T$ uses a valid pattern such as $\mathsf{Any}$ in its input action, and so using our convention of dropping empty provenance annotations from values and valid patterns from input actions, the two systems look the same when expressed in the annotated version. Their transitions, expressed using the provenance tracking reduction relation, change to include provenance as shown below:

$$S \rightarrow_a n\langle\!\langle p : a!\epsilon \rangle\!\rangle \qquad\qquad \begin{aligned} T &\rightarrow_a (\nu m)(m\langle\!\langle p : b!\epsilon \rangle\!\rangle \mid c[m\,(x).n\,\langle x\rangle]) \\ &\rightarrow_a c[n\,\langle p : c?\epsilon\,;\, b!\epsilon\rangle] \\ &\rightarrow_a n\langle\!\langle p : c!\epsilon\,;\, c?\epsilon\,;\, b!\epsilon \rangle\!\rangle \end{aligned}$$

The result of the transitions in each case is a message with channel $n$ and value $p$, similar to what we had in the plain version. The provenance of the value $p$ obtained from system $S$ is different from that obtained from system $T$ however. An observer $O$, equipped with a suitable pattern, would be able to tell these two apart. For instance, using our sample pattern language, the observer $O$ defined as $O \triangleq o[n\,(a!\epsilon \text{ as } x).P]$ would consume the value $p$ when interacting with $S$ but would not consume it when interacting with $T$. The reason is that the provenance of $p$ would match $O$'s policy in the first case, while it would not match it in the second case. This way, the observer $O$ would be able to tell that $S$ and $T$ are in fact different. Not only that, but $O$ can actually detect the names of other principals making up systems $S$ and $T$ and even detect that the value $p$ had originated at $b$ in the case of $T$. An example definition of an observer that would accomplish this is $O \triangleq o[n\,(\text{Any}\,;\, b!\epsilon \text{ as } x).P]$

What this simple example shows is that provenance alters the discriminating power of observers, and as a result changes the behavioural equivalence of the calculus. By checking the provenance of values published by systems, observers are able to discern more differences between systems than would have been otherwise possible. In fact, provenance makes some of the internal behaviour of systems externally visible as we have seen from the example. The remainder of this section is aimed to formalise these points.

To study behavioural equivalences for our provenance calculus, we use as a starting point the intuitive, and increasingly popular, framework of *barbed* bisimulation (and the induced congruence). Barbed bisimulation provides a very natural notion of behavioural equivalence and is easily adaptable to any calculus with a notion of system reduction. We start with the plain version of the calculus, being simpler, and then move on to the annotated version. We contrast the equivalences obtained in the two versions of the calculus, thus highlighting the role that provenance plays in discriminating between systems. Note that the equivalences defined in this section are what is commonly referred to as *weak* equivalences. A weak equivalence is one that abstracts away from the exact number of internal communication steps since these are not usually detectable by an outside observer. In any case, a strong equivalence is often only used as a stepping stone to define the weak equivalence or as an easier to prove notion of equivalence. Neither of these are needed in our case.

### 4.3.1 Plain version

As expected, in the case of the plain version of the calculus, the behavioural equivalences are identical to their counterparts in the asynchronous $\pi$-calculus, save for those changes required to account for the fact that processes are explicitly assigned an identity and for our use of a special version of the input-guarded choice construct. These changes are mostly superficial however, since as we commented several times already, the identities do not play any role in the semantics of the plain version. This fact, we will see, can be formally proved with the behavioural equivalences.

The equivalence we define is called *reduction barbed congruence* and is based on the reduction system of the plain version given in Figure 3.4. It requires that the two systems checked for equivalence have the same basic observables, produce equivalent systems when they reduce and behave in the same way in every context we place them in. As our calculus is based on the asynchronous $\pi$-calculus, only outputs are observable; there is no way, in general, to tell when an input has taken place in an asynchronous setting. Technically, since we split communication into the two separate steps of sending and receiving, messages are what is observable and not output processes. Following common practice in the literature, we only observe the channel name of a message as it is simpler and equivalent to observing the whole message. We give the definitions of the observation predicate and reduction barbed congruence below.

**Definition 4.8** (Plain observation predicate)**.** We say that a channel name $m$ is observable at a system $S$, written $S \downarrow_p m$, if and only if $S \equiv_p (\nu n)(m\langle\!\langle v \rangle\!\rangle \mid S')$ for some channel name $n$, value $v$ and system $S'$ such that $m \neq n$. We also write $S \Downarrow_p m$ to mean that there exists some system $S'$ such that $S \Rightarrow_p S'$ and $S' \downarrow_p m$.

The observation predicate is meant to characterise the *basic observables* (barbs) of a system. When defining these, we need to consider what an outside observer, interacting with the system, can perceive about its behaviour. Since we are in an asynchronous setting, an outside observer can only detect the messages that the system under observation has produced; it cannot tell when the system has consumed a message sent by the observer. The lemma below states that the observable behaviour of a system does not changed when the system is placed in certain contexts. Specifically, the restriction of a channel $n$ does not impact the ability of a system to communicate with the outside world on other channels. Also similarly, placing another system in parallel does not stop a system from being able to communicate on any of its public channels.

**Lemma 4.9.** *The observation predicate is preserved by contexts as described below:*

- $(\nu n)S \downarrow_p m$ *if and only if* $S \downarrow_p m$ *and* $m \neq n$.
- $(S \mid T) \downarrow_p m$ *if and only if* $S \downarrow_p m$ *or* $T \downarrow_p m$.

*Proof.* For each of the two properties, we need to prove both directions of the implication. The proof relies on the definition of the observation predicate and structural congruence to analyse the structure of the system in question. The details are long but straightforward. □

**Definition 4.10** (Plain reduction barbed congruence)**.** Plain (weak) reduction barbed congruence, $\cong_p$, is the largest symmetric binary relation on systems that is:

- barb-preserving: $S \cong_p T$ implies that for all channel names $m$, if $S \downarrow_p m$ then $T \Downarrow_p m$.
- reduction-closed: $S \cong_p T$ implies that if $S \rightarrow_p S'$ for some $S'$ then $T \Rightarrow_p T'$ for some $T'$ such that $S' \cong_p T'$.
- contextual: $S \cong_p T$ implies that $C[S] \cong_p C[T]$ for all contexts $C$.

The three criteria used in the definition of reduction barbed congruence follow naturally from our intuitions of how two equivalent systems are expected to behave. Firstly, two equivalent systems must have the same basic observables, which means they must be able to produce identical messages. They must also reduce to equivalent systems, ensuring that they remain equivalent as they evolve. Finally, when a system is used as a component to build a larger one, we would like to be able to replace it with an equivalent one without changing the behaviour of the larger system. To see how this works, consider again the two systems $S$ and $T$ we saw earlier. We said informally that these two systems would look the same to an external observer and hence they are behaviourally equivalent. We now give a formal proof of their equivalence according to reduction barbed congruence.

**Proposition 4.11.** *Let $S$ and $T$ be the plain systems defined as follows:*

$$S \triangleq a[n\langle p\rangle] \qquad\qquad T \triangleq (\nu m)(b[m\langle p\rangle] \mid c[m(x).n\langle x\rangle])$$

*Systems $S$ and $T$ are reduction barbed congruent in the plain version, that is, it is the case that $S \cong_p T$.*

*Proof.* To prove that $S \cong_p T$ holds, we need to find a relation $r$ that includes $(S, T)$ and exhibits all properties of reduction barbed congruence. Since $\cong_p$ is by definition the largest such relation, it will follow that $r \in \cong_p$ and hence, $S \cong_p T$ will also follow from that. $\square$

Reduction barbed congruence characterises exactly the differences between systems that an external observer would be able to discern by interacting with them. As a result, identities of principals are completely transparent in the plain version of the calculus since this latter does not provide any constructs to interact with them. This means that the behaviour of a system should not be affected by the particular identities we pick for its principals. To make this formal, we define the notion of *renaming*. Renamings are simply substitutions of principal names for principal names. We anticipate that the behaviour of a system is constant under the application of an arbitrary renaming.

**Definition 4.12.** A renaming is a substitution of principal names for principal names.

Renamings are simply a special subset of substitutions. Like substitutions, we use $\sigma, \sigma', \ldots$ to range over them and $S\sigma$ to denote the application of renaming $\sigma$ to system $S$. For example, the application of the renaming $\sigma \triangleq \{^{a',b',c'}/_{a,b,c}\}$ to system $S \triangleq a[P] \mid b[Q] \mid c[R]$ yields the system $S' \triangleq a'[P] \mid b'[Q] \mid c'[R]$. The following lemma, which shows that renamings are preserved by structural congruence and reduction, is needed to prove Theorem 4.14. This latter shows that system equivalence is preserved under renamings, formally proving that principal identities are superfluous in the plain version.

**Lemma 4.13.** *Renamings are preserved by structural congruence and reduction as described below:*

- *if $S \equiv_p T$ then $S\sigma \equiv_p T\sigma$.*
- *if $S \rightarrow_p T$ then $S\sigma \rightarrow_p T\sigma$*

*Proof.* The proof is a straightforward rule induction and hence omitted. $\square$

**Theorem 4.14.** *Let $\sigma$ be a renaming and $S$ be a plain system. It is the case that $S \cong_p S\sigma$.*

*Proof.* Let $r$ be the smallest relation containing pairs of the form $(S, S\sigma)$ and closed under contexts. By definition, $r$ is contextual and hence we only need to show that it preserves observation and is closed under reduction. We proceed by induction on why the pair $(S, T)$ is in $r$, and there are two cases to consider.

**Case 1.** In the first case, $(S, T)$ is of the form $(S', S'\sigma)$. For preservation of observables, assume that $S' \downarrow_p m$. This means that $S' \equiv_p (\nu n)(m\langle\langle v\rangle\rangle \mid S'')$, which implies, by Lemma 4.13, that $S'\sigma \equiv_p ((\nu n)(m\langle\langle v\rangle\rangle \mid S''))\sigma$. This latter simplifies to $(\nu n)(m\langle\langle v\rangle\rangle \mid S''\sigma)$ which means that $S'\sigma \Downarrow_p m$. For reduction closure, assume that $S' \rightarrow_p S''$. Then by Lemma 4.13, this implies that $S'\sigma \rightarrow_p S''\sigma$, and from the first clause in the definition of $r$, we know that $(S'', S''\sigma)$ is in $r$.

**Case 2.** In the second case, $(S, T)$ is of the form $(C[S'], C[T'])$ for some context $C$. We proceed by induction on the structure of $C$. The case $[.]$ holds trivially. Consider the case when $C = (\nu n)[.]$ and assume that $(\nu n)S' \downarrow_p m$. This means, by Lemma 4.9, that $S' \downarrow_p m$ and $m \neq n$. This latter in turn implies, by the induction hypothesis, that $T' \downarrow_p m$, which again implies, by Lemma 4.9, that $(\nu n)T' \downarrow_p m$. The case for parallel composition follows in a similar manner using the second clause of Lemma 4.9. $\square$

## 4.3.2 Annotated version

We can adapt the definition of reduction barbed congruence to the annotated version of the calculus with practically no changes. However, the equivalence we obtain is different in its discriminating power. As we have already commented informally, provenance exposes principal names and some of the internal behaviour of systems to external observers. Hence, observers are able to detect more differences between systems in the annotated version of the calculus compared to the plain version. The definitions of the annotated versions of the observation predicate and reduction barbed congruence are given below.

**Definition 4.15** (Annotated observation predicate)**.** We say that a channel name $m$ is observable at a system $S$, written $S \downarrow_a m$, if and only if $S \equiv_a (\nu n)(m\langle\langle v\rangle\rangle \mid S')$ for some name $n$, value $v$ and system $S'$ such that $m \neq n$. We also write $S \Downarrow_a m$ to mean that there exists some system $S'$ such that $S \Rightarrow_a S'$ and $S' \downarrow_a m$.

**Definition 4.16** (Annotated reduction barbed congruence)**.** Annotated (weak) reduction barbed congruence, $\cong_a$, is the largest symmetric binary relation on systems that is:

- barb-preserving: $S \cong_a T$ implies that for all names $m$, if $S \downarrow_a m$ then $T \Downarrow_a m$.

- reduction-closed: $S \cong_a T$ implies that if $S \rightarrow_a S'$ for some $S'$ then $T \Rightarrow_a T'$ for some $T'$ such that $S' \cong_a T'$.
- contextual: $S \cong_a T$ implies that $C[S] \cong_a C[T]$ for all contexts $C$.

The definition of annotated reduction barbed congruence looks the same as the plain one, albeit modified to use the annotated versions of observation, reduction and contexts. The extra discriminating power of the equivalence comes primarily from the ability of contexts to check the provenance of values they receive from the system; these reveal details about the system that the observer would not have had access to in the plain version. To see this in action, consider again our example systems $S \triangleq a[n\langle p\rangle]$ and $T \triangleq (\nu m)(b[m\langle p\rangle] \mid c[m(x).n\langle x\rangle])$. Neither of these has any immediate observables and so the first criteria of equivalence, barb preservation, is satisfied. Recall that $S$ has only one reduction yielding $n\langle\!\langle p : a!\epsilon\rangle\!\rangle$. Recall also that $T$ reduces to a similar system $n\langle\!\langle p : c!\epsilon \,;\, c?\epsilon \,;\, b!\epsilon\rangle\!\rangle$ in three reduction steps. From the definition of the observation predicate, we can show that both $n\langle\!\langle p : a!\epsilon\rangle\!\rangle \downarrow_a n$ and $n\langle\!\langle p : c!\epsilon \,;\, c?\epsilon \,;\, b!\epsilon\rangle\!\rangle \downarrow_a n$ hold. The second criteria of equivalence, reduction closure, allows for a single reduction step from one system to be matched by multiple steps from the other system. Therefore, $S$ and $T$ satisfy the reduction-closure criteria. Now let us check if they satisfy the third and final criteria, that is, we want to check whether $S$ and $T$ remain equivalent when placed in an arbitrary context. It turns out this is not the case, and we have already hinted at a context that is capable of distinguishing between them. The observer $O \triangleq o[n(x \text{ as } a!\epsilon).P]$ which we argued informally may be used to tell $S$ and $T$ apart, can be turned into the context $C \triangleq [.] \mid o[n(x \text{ as } a!\epsilon).eureka\langle eureka\rangle]$, where $eureka\langle eureka\rangle$ is just an output action used by observers to declare the successful consumption of a particular message. Using this context, we find that $C[S] \Downarrow_a eureka$ holds. However, $C[T] \Downarrow_a eureka$ does not hold as the provenance of $p$ produced by $T$ does not match the pattern $a!\epsilon$ and hence the context $C$ would not be able to consume $T$'s message and declare $eureka$ as it does in the case of $S$. Hence, we can conclude that $S$ and $T$ are not reduction barbed congruent in the annotated version of the calculus. This is formally proved in Proposition 4.17. We also prove that the renaming property does not hold for the annotated version. The details can be found in Theorem 4.18.

**Proposition 4.17.** *Let $S$ and $T$ be the annotated systems defined as follows:*

$$S \triangleq a[n\langle p\rangle] \qquad\qquad T \triangleq (\nu m)(b[m\langle p\rangle] \mid c[m(x).n\langle x\rangle])$$

*Systems $S$ and $T$ are not reduction barbed congruent in the annotated version, that is, it is not the case that $S \cong_a T$.*

*Proof.* The discussion above contains most of the details of this proof. Although $S$ and $T$ satisfy the barb preservation and reduction closure properties of the equivalence, they fail to satisfy the contextuality property. A context that demonstrates this is $C \triangleq [.] \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle]$. Neither $C[S]$ nor $C[T]$ has any immediate observables. However, after some reduction steps, it is possible for $C[S]$ to exhibit an observable that $C[T]$ cannot match as shown below:

$$\begin{aligned}
C[S] &= a[n\,\langle p\rangle] \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\to_a n\langle\!\langle p : a!\epsilon\rangle\!\rangle \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\to_a o[eureka\,\langle eureka\rangle] \\
&\to_a eureka\langle\!\langle eureka\rangle\!\rangle
\end{aligned}$$

$$\begin{aligned}
C[T] &= (\nu m)(b[m\,\langle p\rangle] \mid c[m\,(x).n\,\langle x\rangle]) \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\to_a (\nu m)(m\langle\!\langle p : b!\epsilon\rangle\!\rangle \mid c[m\,(x).n\,\langle x\rangle]) \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\to_a c[n\,\langle p : c?\epsilon\,;\, b!\epsilon\rangle] \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\to_a n\langle\!\langle p : c!\epsilon\,;\, c?\epsilon\,;\, b!\epsilon\rangle\!\rangle \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle] \\
&\not\to_a
\end{aligned}$$

From the above, we can infer that $C[S] \not\cong_a C[T]$, and hence conclude that $S \not\cong_a T$. □

**Theorem 4.18.** *Let $\sigma$ be a renaming and $S$ be an annotated system. It is not the case that $S \cong_a S\sigma$.*

*Proof.* Even in the annotated version, there are special cases where the renaming property holds. What the above theorem says is that renaming does not hold in general. To prove this, we just need to give an example where the application of a renaming alters the observable behaviour of a system. For instance, consider the very simple system $a[m\,\langle p\rangle]$. Any renaming that alters the name of principal $a$ would be recognisable by an outside observer. Take for example the renaming $\{\{^b/_a\}\}$. Applied to system $a[m\,\langle p\rangle]$, it yields $b[m\,\langle p\rangle]$. The two systems $a[m\,\langle p\rangle]$ and $b[m\,\langle p\rangle]$ are not reduction barbed congruent and to distinguish between them we can use the same context we used in the proof of Proposition 4.17, that is, context $C \triangleq [.] \mid o[n\,(x \text{ as } a!\epsilon).eureka\,\langle eureka\rangle]$. This context would consume value $p$ in the case of $a[m\,\langle p\rangle]$ but not in the case of $b[m\,\langle p\rangle]$. Hence it would be able to declare *eureka* in the case of the original system while it would not after the application of the renaming. □

**Theorem 4.19.** *$S \cong_a T$ implies $|S| \cong_p |T|$*

*Proof.* Let us assume that $S \cong_a T$ holds. We need to prove that $|S| \cong_p |T|$ follows from this. From the assumption, we can deduce that $S$ and $T$ have the same basic observables, reduce to equivalent systems and the systems obtained by placing them in a context are also equivalent. We just need to show that these properties imply that their counterparts in the plain version hold. $\square$

**Definition 4.20.** Let $\cong_{ppt}$ denote the relation defined like $\cong_a$ but with the reduction relation $\rightarrow_a$ replaced with $\rightarrow_{ppt}$. We refer to $\cong_{ppt}$ as *pure* provenance tracking reduction barbed congruence, or as pure provenance tracking congruence for short.

**Theorem 4.21.** *$S \cong_{ppt} T$ if and only if $|S| \cong_p |T|$.*

*Proof.* We need to prove both directions of the implication. The direction $S \cong_{ppt} T$ implies $|S| \cong_p |T|$ follows from Theorem 4.19 as $\cong_{ppt}$ is contained in $\cong_a$ (a result of $\rightarrow_{ppt}$ being a special case of $\rightarrow_a$). $\square$

## 4.4 Concluding remarks

By making the calculus parametric on the pattern matching language, we are able to prove generic results that apply to all pattern languages or to a particular subset of them. We are also able to more easily study different notions of provenance by simply instantiating the calculus with a suitable pattern language. The pattern language provides the interface through which principals access provenance. We saw that pure provenance tracking does not alter the computational semantics of the calculus. It is only when principals are allowed to make decisions that depend on the provenance of values that the computational semantics of the calculus changes. The change in our case consisted of prohibiting, through dynamic provenance checking, input transitions where the provenance of data does not comply with the policies of principals. This was demonstrated in a number of formal results that compared the reduction relations of the plain and annotated versions, and it allowed us to highlight the role of provenance in the calculus.

Comparing the reduction semantics of the plain and annotated versions of the calculus was not the only method we used to study the role of provenance in the calculus. In addition to this, we also studied behavioural equivalences for both versions of the calculus and contrasted their discriminating power. We found that provenance gives observers the ability to discern more differences between systems than is possible in the plain version. This is of course subject to the expressive power of the pattern matching language. If the pattern matching language suppressed all information about provenance from observers, then the resulting equivalence would be the same as that of the plain version. This, and other results were proved formally, further strengthening our results.

The behavioural equivalences we developed are based on barbed congruence. Although very intuitive and easy to understand, they rely in their definition on a universal quantification over contexts, making them hard to use in general to prove the equivalence of systems. To alleviate this, we have developed characterisations of them based on the powerful coinductive notion of bisimulation. The bisimulation equivalences are defined on top of labelled transition systems and coincide with the contextually defined equivalences, hence providing a sound and complete proof methodology for showing the equivalence of systems in every context. The definitions of the labelled transition systems, the bisimulation equivalences and full abstraction results are all given in Appendix A.

# Chapter 5

# Semantics of Provenance

In the previous two chapters, we presented our provenance calculus and described its main features, namely those of annotated data, provenance tracking and provenance queries. Our presentation of the calculus was in the form of two versions, a plain version and an annotated version. Contrasting the two versions allowed us to highlight the role of provenance tracking and checking in the calculus. We elaborated on this further by defining behavioural equivalences for both versions of the calculus and comparing their discriminating power. In this chapter, we turn our attention to the problem of defining the semantics of provenance sequences and formalising some of their properties.

We start by noting that not all syntactically legal provenance sequences are semantically meaningful. We have already alluded to this briefly in Section 4.1.2 with the example of provenance sequence $a!\epsilon \,;\, a!\epsilon \,;\, \epsilon$. There, we argued informally that the provenance of a value is meant to be a record of some past behaviour, and as such, the system containing the value must be the result of some sequence of transitions obeying the rules of the provenance tracking reduction relation. Since no sequence of system transitions following the rules of our calculus would ever give rise to provenance of the form $a!\epsilon;a!\epsilon;\epsilon$, we can then conclude that such a provenance sequence cannot be semantically meaningful. This intuitive idea leads us to define the first property of provenance sequences, that of *well-formedness*. A well-formed system is one that may be derived from an *initial* system by a sequence of zero or more transitions. To understand what initial systems are, note that provenance is a run time property, meaning that associating provenance with values only makes sense at run time, and hence in their default static state, systems should only have empty provenance. Any system with non-empty provenance is meant to represent a run time system that is the result of transitions starting from some initial state. The definition of well-formedness we give is a semantic one relying on the ability to derive what may be a very long sequence of transitions. To alleviate this, we define a syntactic characterisation of well-formedness, which is much easier to use to prove well-formedness.

Well-formedness is a very weak property though, as all it guarantees about a system is that there exists some sequence of transitions that leads to it, not that this sequence of transitions is what really took place. We call this stronger property soundness or correctness. In order to give a formal definition of soundness, we present a first-order temporal logic whose modalities are interpreted with respect to the past of the system.[1] We then use this to define the denotation of provenance sequences and the notion of sound provenance. Based on these definitions, we prove that our provenance tracking reduction relation always produces correct provenance sequences. In other words, we show that provenance sequences produced by the provenance tracking reduction relation "cannot lie" or "cannot be forged". These properties of provenance are especially important if we consider more complicated notions of provenance or settings where we allow principals to control provenance tracking.

It should be noted that the properties of the calculus, and indeed the calculus itself, rely on the assumption that the underlying infrastructure is trusted. By infrastructure here we mean the combination of software and hardware components that implement the communication and provenance management capabilities of the calculus. This infrastructure is assumed to allow principals to communicate by message passing, to keep track of the provenance of data thus communicated, and to make this provenance available to principals via pattern matching, all as prescribed by the semantics of the calculus in Chapter 3.

## 5.1 Well-formed Provenance

Provenance sequences are aimed to represent some past system behaviour. As such, not all provenance sequences can be considered well-formed as there are some that cannot arise from any legal system behaviour. For example, the provenance sequence $a!\kappa;b!\kappa';\kappa''$, which is just a generalisation of the sequence $a!\epsilon;a!\epsilon;\epsilon$ we already looked at, is not well-formed because it tells us that its value was sent by $a$ and immediately before that it was sent by $b$. This, however, is impossible according to our provenance tracking semantics as send and receive events always interleave. Moreover, some provenance sequences may be well-formed in certain contexts but not others. For example, consider the provenance sequence $a?\kappa\,;\,\kappa'$. The only context where this provenance would make

[1] As opposed to the more prevalent uses of modalities in temporal logics which usually refer to the future.

sense is if it occurs as the provenance of a value located at principal $a$. Indeed, it is impossible, based on the semantics of our calculus, to produce a value with such a provenance at any principal other than $a$. We start by formalising these intuitions in the definition of well-formed systems. We then present a sound and complete syntactic characterisation of it.

### 5.1.1 Definition of well-formedness

The simplest class of well-formed systems is that of initial systems, whose definition is given below. These can be thought of as representing the static state of systems before they have made any transitions. As such, initial systems contain no messages as these are considered values in transit and are only present in a system after a principal makes a send action. In addition to this, initial systems only contain values with empty provenance; non-empty provenance only arises after a system performs one or more transitions.

**Definition 5.1.** A system $S$ is *initial* if it contains no messages and all its values have empty provenance.

As shown by the below proposition, name restriction, parallel composition and structural congruence do not affect whether a given system is initial or not.

**Proposition 5.2.** *The following properties hold for initial systems:*

- *$S$ is initial if and only if $(\nu n)S$ is initial.*
- *$S$ and $T$ are initial if and only if $S \mid T$ is initial.*
- *If $S$ is initial and $S \equiv_a S'$ then $S'$ is initial.*

*Proof.* All three properties are easy to prove since none of the transformations changes the provenance of values in a system or introduces messages into it, and hence an initial system would remain so under any combination of these transformations. □

The definition of well-formed systems, that is systems whose values all have well-formed provenance, is then straightforward. A system is considered well-formed if it is reachable from an initial system by a sequence of zero or more transitions.

**Definition 5.3.** A system $S$ is *well-formed* if there exists an initial system $S_0$ such that $S_0 \Rightarrow_a S$.

Already from this definition we can deduce a number of properties of well-formed systems. For example, an initial system is well-formed according to this definition since it can be derived from itself using a zero-length transition. Another property that we may deduce is that of preservation of well-formedness under reduction. That is, starting from a well-formed system, our reduction relation is guaranteed to only produce well-formed systems. In fact, we can deduce that well-formedness is also preserved under the same three system transformations as initiality. The properties of well-formedness are summarised by the below proposition.

**Proposition 5.4.** *The following properties hold for well-formed systems:*

- *If $S$ is initial then $S$ is well-formed.*
- *If $S$ is well-formed and $S \rightarrow_a S'$ then $S'$ is well-formed.*
- *If $S$ is well-formed and $S \equiv_a S'$ then $S'$ is well-formed.*
- *If $S$ is well-formed then so is $(\nu n)S$.*
- *If $S$ and $T$ are well-formed then so is $S \mid T$.*

*Proof.* All five properties are easy to prove. The first one follows simply from the definition of $\Rightarrow_a$ since for any system $S$ it holds that $S \Rightarrow_a S$. Hence, if $S$ is initial then it follows from the definition of well-formedness that $S$ is well-formed. For the second property, that is $S$ is well-formed and $S \rightarrow_a S'$ implies $S'$ is well-formed, we start by assuming that $S$ is well-formed and that $S \rightarrow_a S'$. By definition, $S$ being well-formed implies that there exists an initial system $S_0$ such that $S_0 \Rightarrow_a S$. This means that we have $S_0 \Rightarrow_a S \rightarrow_a S'$, from which it follows that $S_0 \Rightarrow_a S'$. Hence, $S'$ is well-formed. The last three properties follow from the rules of the reduction relation for structural congruence, restriction and parallel composition respectively. □

### 5.1.2 Syntactic characterisation of well-formedness

The definition of well-formedness given above is a semantic one. To prove that a system is well-formed according to this definition, we need to show that it is derivable from an initial system using a sequence of zero or more transitions. However, this may not be easy if the system in question is large and complex. What is needed then is an alternative characterisation of well-formedness that only relies on the syntactic analysis of the system itself. We do this by defining the following set of judgements:

- $S \vdash \diamond$: System $S$ is well-formed.
- $P \vdash_a \diamond$: Process $P$ is well-formed at principal $a$.
- $w \vdash_a \diamond$: Annotated value $w$ is well-formed at principal $a$.
- $\kappa \vdash_a \diamond$: Provenance $\kappa$ is well-formed at principal $a$.

These judgements are defined inductively on the structure of systems, processes, values and provenance sequences respectively. The full definition is given in Figure 5.1.

Figure 5.1. *Syntactic characterisation of well-formedness*

$$\textsc{SWF Nil}\quad \frac{}{0 \vdash \diamond} \qquad \textsc{SWF Prin}\quad \frac{P \vdash_a \diamond}{a[P] \vdash \diamond} \qquad \textsc{SWF Msg}\quad \frac{\kappa \vdash_a \diamond \quad \kappa' \vdash_a \diamond}{m\langle\!\langle v : a!\kappa\, ; \kappa' \rangle\!\rangle \vdash \diamond} \qquad \textsc{SWF Res}\quad \frac{S \vdash \diamond}{(\nu n)S \vdash \diamond} \qquad \textsc{SWF Par}\quad \frac{S \vdash \diamond \quad T \vdash \diamond}{S \mid T \vdash \diamond}$$

$$\textsc{PWF Out}\quad \frac{w \vdash_a \diamond \quad w' \vdash_a \diamond}{w\,\langle w' \rangle \vdash_a \diamond} \qquad \textsc{PWF In}\quad \frac{w \vdash_a \diamond \quad P_i \vdash_a \diamond \text{ for all } i \text{ in } I}{\Sigma_{i \in I} w\,(\pi_i \;\mathsf{as}\; x).P_i \vdash_a \diamond} \qquad \textsc{PWF Res}\quad \frac{P \vdash_a \diamond}{(\nu n)P \vdash_a \diamond}$$

$$\textsc{PWF Par}\quad \frac{P \vdash_a \diamond \quad Q \vdash_a \diamond}{P \mid Q \vdash_a \diamond} \qquad \textsc{PWF If}\quad \frac{w \vdash_a \diamond \quad w' \vdash_a \diamond \quad P \vdash_a \diamond \quad Q \vdash_a \diamond}{\mathsf{if}\; w = w' \;\mathsf{then}\; P \;\mathsf{else}\; Q \;\vdash_a \diamond}$$

$$\textsc{PWF Rep}\quad \frac{P \vdash_a \diamond \quad \text{all bound names in } P \text{ have empty provenance}}{*P \vdash_a \diamond}$$

$$\textsc{VWF Var}\quad \frac{}{x \vdash_a \diamond} \qquad \textsc{VWF Val}\quad \frac{\kappa \vdash_a \diamond}{v : \kappa \vdash_a \diamond} \qquad \textsc{PWF Eps}\quad \frac{}{\epsilon \vdash_a \diamond} \qquad \textsc{PWF In}\quad \frac{\kappa \vdash_a \diamond \quad \kappa' \vdash_b \diamond \quad \kappa'' \vdash_b \diamond}{a?\kappa\, ; b!\kappa'\, ; \kappa'' \vdash_a \diamond}$$

The empty system 0 is initial and hence well-formed. The system $a[P]$ is well-formed if the process $P$ is well-formed in the context where it appears, that is at principal $a$. A message is well-formed only if the provenance of its value is of the form $a!\kappa\, ; \kappa'$, for some provenance sequences $\kappa$ and $\kappa'$ that are well-formed at principal $a$. To see why this is the case, consider that a message represents a value that has been sent by some principal in a previous transition and that has not yet been consumed. Its provenance must represent this. The provenance sequences $\kappa$ and $\kappa'$ denote the provenance of the channel name and the previous provenance of the value respectively. Both of these were located at $a$ prior to the send action that produced the message, and as such, these two provenance sequences must be well-formed at $a$. Restriction and parallel composition simply state that well-formedness is indeed preserved by these two operations.

For processes, all we need to do is guarantee that all values that appear in the process are well-formed at the principal. The only rule of note is that for replication, which states that all bound names must have empty provenance. The reason is that for a bound name to participate in communication, and hence for it to have non-empty provenance, we need to unfold any replication surrounding it. Once the replication is unfolded, it cannot be folded again with the rest of the replication as any non-empty provenance it has would mean it would not match the replicated process. To illustrate this, consider the following initial system:

$$S \triangleq a[*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid b[m{:}\epsilon\,(x).P] \mid c[m{:}\epsilon\,(y).Q]$$

which is composed of three principals $a$, $b$ and $c$. Principal $a$ contains the replicated process $*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)$ and therefore is able to continuosly generate and send a fresh value $n$ on channel $m$. Principals $b$ and $c$ both listen for data sent on channel $m$. As this is an initial system, all values have empty provenance. This system has only one transition; principal $a$ could send a fresh value $n$ on channel $m$. However, the only way for this to take place is by applying the structural congruence rule $a[*P] \equiv_a a[P \mid *P]$ first. This rule allows the replication to unfold, making it possible to apply one of the rules of reduction. In the case of system $S$, the structural congruence rule for replication allows us to replace $a[*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)]$ with $a[(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle) \mid *(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)]$. Applying another rule of structural congruence, namely $a[P \mid Q] \equiv_a a[P] \mid a[Q]$, allows us to replace $a[(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle) \mid *(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)]$ with $a[(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid a[*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)]$. Therefore, the following holds:

$$S \equiv_a a[(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid a[*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid b[m{:}\epsilon\,(x).P] \mid c[m{:}\epsilon\,(y).Q]$$

We still need to extrude the scope of the bound name $n$ (the one not under replication as we do not have any rules that allow scope extrusion from under a replication directly). Doing this gives us:

$$S \equiv_a (\nu n)(a[(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)]) \mid a[*(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid b[m{:}\epsilon\,(x).P] \mid c[m{:}\epsilon\,(y).Q]$$

Now, we are able to apply the reduction rule for send to get:

$$S \rightarrow_a (\nu n)(m\langle\!\langle n{:}a!\epsilon\,;\,\epsilon\rangle\!\rangle) \mid a[*\,(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid b[m{:}\epsilon\,(x).P] \mid c[m{:}\epsilon\,(y).Q]$$

Note how the replicated process was unfolded to give us a principal that is able to make a send action. The provenance of the value changed after the send action. However, the replicated process is still the same and the value $n$ still has empty provenance. By applying more rules of structural congruence and reduction, we are able to derive the following transition:

$$S \Rightarrow_a a[*\,(\nu n)(m{:}\epsilon\,\langle n{:}\epsilon\rangle)] \mid (\nu n)(b[P\{^{n:b?\epsilon;a!\epsilon;\epsilon}/_{x}\}]) \mid (\nu n)(c[Q\{^{n:c?\epsilon;a!\epsilon;\epsilon}/_{y}\}]) \mid$$

The replicated process at principal $a$, and especially the bound value $n$ under the replication will always have empty provenance as it cannot be involved in any communication unless the replication is unfolded as shown in the example.

There are two rules for values, one for variables which simply states that all variables are well-formed since they have no provenance, and one for annotated values which requires the provenance of the value to be well-formed at the same principal.

For provenance sequences, only two forms are well-formed at a principal, the empty provenance for values that originated at the principal itself and that have not participated in any transitions, and provenance sequences of the form $a?\kappa\,;\,b!\kappa'\,;\,\kappa''$, which is used for values that a principal has received from another one. In this latter case, the provenance of the receive channel must itself be well-formed at the same principal, while the provenance of the send channel and the previous provenance of the value must be well-formed at the principal that sent the value.

Having defined the well-formedness judgements, Theorem 5.5 shows that this characterisation of well-formedness does indeed coincide with the previous semantically defined notion of well-formedness.

**Theorem 5.5** (Well-formedness - Soundness and Completeness)**.** *$S \vdash \diamond$ if and only if $S$ is well-formed.*

*Proof.* We have to show that both directions of the implication hold, that is if $S \vdash \diamond$ then there exists an initial system $S_0$ such that $S_0 \Rightarrow_a S$ (i.e. $S \vdash \diamond$ is sound), and that if $S_0 \Rightarrow_a S$ for some initial system $S_0$ then $S \vdash \diamond$ (i.e. $S \vdash \diamond$ is complete).

**Soundness.** We need to show that if $S \vdash \diamond$ then there exists an initial system $S_0$ such that $S_0 \Rightarrow_a S$. Let us assume that $S \vdash \diamond$ holds. We proceed by induction on its derivation. There are five cases as follows:

Case SWF Nil. This means that $S = 0$ and hence it is clearly well-formed as it is an initial system.

Case SWF Prin. This means that $S$ is of the form $a[P]$ for some principal name $a$ and process $P$. Hence, the derivation of $S \vdash \diamond$ must have been of the form $\dfrac{P \vdash_a \diamond}{a[P] \vdash \diamond}$. Upon analysis of the rules of Figure 5.1, it should be clear that all values in $P$ must be in one of the following three forms:

$$x \qquad\qquad v : \epsilon \qquad\qquad v : a?\kappa_1 \,;\, b!\kappa_2 \,;\, \kappa_3$$

If no value in $P$ has the third form above, then $a[P]$ is initial and hence well-formed, and the proof is done. If, on the other hand, at least one value in $P$ has the form $v{:}a?\kappa_1;b!\kappa_2;\kappa_3$, then we can show that $v : a?\kappa_1 \,;\, b!\kappa_2 \,;\, \kappa_3 \vdash_a \diamond$ must hold. This latter must have been derived using PWF In, and so we can infer that $\kappa_1 \vdash_a \diamond$, $\kappa_2 \vdash_b \diamond$, and $\kappa_3 \vdash_b \diamond$ all hold.

We can show that $P \equiv_a P'\{^{v:a?\kappa_1;b!\kappa_2;\kappa_3}/_x\}$ if $v \notin bn(P)$ and that $P \equiv_a (\nu v)(P'\{^{v:a?\kappa_1;b!\kappa_2;\kappa_3}/_x\})$ if $v \in bn(P)$. Note that this is only possible because we know that all bound names appearing under a replication in $P$ must have empty provenance as required by rule PWF Rep.

Let us consider the first case, that is when $P \equiv_a P'\{^{v:a?\kappa_1;b!\kappa_2;\kappa_3}/_x\}$. Now, take the following system:

$$T = m\langle\!\langle v : b!\kappa_2 \,;\, \kappa_3\rangle\!\rangle \mid a[m{:}\kappa_1(x).P']$$

We have $T \rightarrow_a S$ by rule ARed Rcv, and given that $\kappa_1 \vdash_a \diamond$, $\kappa_2 \vdash_b \diamond$, and $\kappa_3 \vdash_b \diamond$ hold then we can show that $T \vdash \diamond$. Therefore, by the induction hypothesis, $T$ is well-formed and hence so is $S$.

The case when $P \equiv_a (\nu v)(P'\{^{v:a?\kappa_1;b!\kappa_2;\kappa_3}/_x\})$ proceeds similarly.

Case SWF Msg. This means that $S$ is of the form $m\langle\!\langle v : a!\kappa_1 \,;\, \kappa_2\rangle\!\rangle$ and so the derivation of $S \vdash \diamond$ must have been of the form:

$$\frac{\kappa_1 \vdash_a \diamond \qquad \kappa_2 \vdash_a \diamond}{m\langle\!\langle v : a!\kappa_1 \,;\, \kappa_2\rangle\!\rangle}$$

Consider the system $a[m : \kappa_1 \langle v : \kappa_2\rangle]$. Using rule ARed Snd, we get:

$$a[m : \kappa_1 \langle v : \kappa_2\rangle] \rightarrow_a m\langle\!\langle v : a!\kappa_1 \,;\, \kappa_2\rangle\!\rangle$$

Since $\kappa_1 \vdash_a \diamond$ and $\kappa_2 \vdash_a \diamond$ hold as shown above, then it is possible to show that $a[m : \kappa_1 \langle v : \kappa_2 \rangle] \vdash \diamond$ holds. From this, and the induction hypothesis, we can infer that $a[m : \kappa_1 \langle v : \kappa_2 \rangle]$ is well-formed, and therefore we conclude that $m\langle\!\langle v : a!\kappa_1 \,;\, \kappa_2 \rangle\!\rangle$ is well-formed.

Case SWF Res. This means that $S$ is of the form $(\nu n)S'$ and so the derivation of $S \vdash \diamond$ must have been of the form:

$$\frac{S' \vdash \diamond}{(\nu n)S' \vdash \diamond}$$

By the induction hypothesis, $S'$ is well-formed and therefore we conclude that $(\nu n)S'$ is well-formed.

Case SWF Par. This means that $S$ is of the form $S' \mid S''$ and so the derivation of $S \vdash \diamond$ must have been of the form:

$$\frac{S' \vdash \diamond \qquad S'' \vdash \diamond}{S' \mid S'' \vdash \diamond}$$

By the induction hypothesis, $S'$ and $S''$ are well-formed and therefore we conclude that $S' \mid S''$ is well-formed.

**Completeness.** We need to show that if $S_0 \Rightarrow_a S$ for some initial system $S_0$ then $S \vdash \diamond$ holds. Let us assume that $S_0 \Rightarrow_a S$ holds. We proceed by induction on the length of this transition. The case when the length of the transition is 0 holds vacuously. In the induction case, the length of the transition is $n + 1$ for some $n \geq 0$. That is, the transition can be written as $S_0 \Rightarrow_a S_n \rightarrow_a S$ for some system $S_n$ where the transition $S_0 \Rightarrow_a S_n$ is of length $n$. By the induction hypothesis, it holds that $S_n \vdash \diamond$. We show that $S \vdash \diamond$ by induction on the derivation of the transition $S_n \rightarrow_a S$. There are seven cases as defined in Figure 3.6. We illustrate a few of them below.

Case ARed Snd. This means the transition $S_n \rightarrow_a S$ was of the form $a[m{:}\kappa_m \langle v{:}\kappa_v \rangle] \rightarrow_a m\langle\!\langle v : a!\kappa_m \,;\, \kappa_v \rangle\!\rangle$. Since $a[m{:}\kappa_m \langle v{:}\kappa_v \rangle] \vdash \diamond$ then from SWF Prin we have $\kappa_m \vdash_a \diamond$ and $\kappa_v \vdash_a \diamond$. This implies by application of SWF Msg that $m\langle\!\langle v : a!\kappa_m \,;\, \kappa_v \rangle\!\rangle \vdash \diamond$.

Case ARed Rcv. This means that the transition $S_n \rightarrow_a S$ was of the form:

$$\frac{j \in I \qquad \kappa_p \models \pi_j}{a[\Sigma_{i \in I} m{:}\kappa_m\, (\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p{:}\kappa_p \rangle\!\rangle \rightarrow_a a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

We know from the induction hypothesis that $a[\Sigma_{i \in I} m{:}\kappa_m\, (\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p{:}\kappa_p \rangle\!\rangle \vdash \diamond$ holds. From this, we can show that $\kappa_m \vdash_a \diamond$ and $P_i \vdash_a \diamond$ for all $i \in I$ hold, and that $\kappa_p$ must be of the form $b!\kappa_1 \,;\, \kappa_2$. From these, it can be shown that $a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}] \vdash \diamond$ indeed holds.

Case ARed If$_t$. This means that the transition $S_n \rightarrow_a S$ was of the form:

$$\frac{}{a[\textsf{if } m{:}\kappa = m{:}\kappa' \textsf{ then } P \textsf{ else } Q\ ] \rightarrow_a a[P]}$$

We know from the induction hypothesis that $a[\textsf{if } m{:}\kappa = m{:}\kappa' \textsf{ then } P \textsf{ else } Q\ ] \vdash \diamond$. From this, we can show that, among other things, $P \vdash_a \diamond$. Hence, we conclude that $a[P] \vdash \diamond$.

Case ARed If$_f$ is similar to the above. The other three cases, ARed Res, ARed Par, and ARed Str can all be shown to hold in a similar fashion. □

Before we end this section, it is worth noting that some aspects of well-formedness may be captured in the calculus at the syntax level by modifying the grammars used in the calculus. However, accounting for all the requirements of well-formedness that way would become very tedious as we would need to incorporate all the conditions found in the rules of Figure 5.1. This would only get more complicated if we were to consider more sophisticated notions of provenance. Hence, it is better to keep well-formedness as a separate semantic notion as given Definition 5.3, and aim to characterise it syntactically like we did in this section if needed.

## 5.2 A logic for the past

In this section, we present a simple logic, the aim of which is to act as a target for defining the semantics of provenance sequences. What we mean by this is that our goal is to be able to define the semantics of provenance sequences as formulae in this logic. Consequently then, the properties that we would like to be able to express in the logic are meant to mirror those captured by provenance. Therefore, like provenance, the formulae of the logic are interpreted with respect to the past, meaning that their truth is judged relative to the past states of systems. Technically, the logic is an extension of first-order logic with the modality $<\alpha>A$, which intuitively means that action $\alpha$ took place *sometime in the past*, and that before action $\alpha$ took place, formula $A$ was true. The remainder of this section defines the syntax and semantics of the logic.

### 5.2.1 Logical formulae

The syntax of the logic is given in Figure 5.2. We let $A, B, \ldots$ range over formulae, $\alpha, \beta, \ldots$ range over actions and $x, y, \ldots$ range over variables. We use **T** for the constant true, $\neg A$ for the negation of $A$, and $A \wedge B$ for the conjunction of $A$ and $B$. We denote existential quantification by $\exists x.A$ where the variable $x$ ranges over channel names. We also use $<\alpha>A$ to denote the sometime in the past modality, which as we already mentioned, can be understood to mean that action $\alpha$ occurred at some point in the past, before which $A$ held true.

There are four types of actions:

- the output action $a.\mathsf{snd}(m, p)$ which denotes that principal $a$ sent value $p$ on channel $m$,
- the input action $a.\mathsf{rcv}(m, p)$ which denotes that principal $a$ received value $p$ on channel $m$,
- the if-true action $a.\mathsf{ift}(p, q)$ which says that principal $a$ tested $p$ and $q$ for equality and that resulted in true, and
- the if-false action $a.\mathsf{iff}(p, q)$ which says that principal $a$ tested $p$ and $q$ for equality and that returned false.

The constant false $\mathbf{F}$, disjunction $\vee$, implication $\Rightarrow$, double implication $\Leftrightarrow$, and the universal quantifier $\forall$ may be defined in terms of $\mathbf{T}$, $\neg$ and $\wedge$, $\exists$ in the standard way. As usual, the quantifiers are binders and hence the variable $x$ in formulae $\exists x.A$ and $\forall x.A$ is bound with scope $A$. This leads to the standard definitions of bound and free variables of a formula $A$, which we denote by $fv(A)$ and $bv(A)$ respectively. We use $A\{^m/_x\}$ for the capture avoiding substitution of $m$ for the free occurrences of $x$ in $A$.

Figure 5.2. *The provenance logic: formulae*

| $A, B ::=$ | logic formulae | $\alpha,\beta ::=$ | actions |
|---|---|---|---|
| $\mathbf{T}$ | constant true | $a.\mathsf{snd}(p, q)$ | output action |
| $\neg A$ | negation of $A$ | $a.\mathsf{rcv}(p, q)$ | input action |
| $A \wedge B$ | conjunction of $A$ and $B$ | $a.\mathsf{ift}(p, q)$ | if-true action |
| $<\alpha>A$ | sometime in the past modality | $a.\mathsf{iff}(p, q)$ | if-false action |
| $\exists x.A$ | existential quantification | | |

### 5.2.2 Satisfaction

The semantics of logical formulae is defined in terms of the satisfaction relation $s \models A$ where $s$ is a *log* and $A$ is a formula. Logs, ranged over by $s, t, \ldots$, are sequences of actions that record the past transitions of a system. Their syntax is given in Figure 5.3. We use $\epsilon$ for the empty sequence, $\alpha$ for the singleton sequence containing action $\alpha$, and $s\,;t$ for the concatenation of sequences $s$ and $t$. As usual, concatenation is associative but not commutative and has $\epsilon$ as identity element. We wish to point out here that despite their similarity to traces, logs are meant as records of *the entire past* of a system, not just its *observable* behaviour. This is needed as the provenance tracking semantics does indeed record actions which would otherwise be unobservable to an external observer.

Figure 5.3. *Syntax of logs*

| | |
|---|---|
| $s,t ::=$ | logs |
| $\epsilon$ | empty sequence |
| $\alpha$ | singleton sequence |
| $s\,;t$ | concatenation |

For a given log $t$, we let $length(t)$ denote its length and $t_{i..j}$, where $i$ and $j$ are natural numbers such that $i \leq j$ and $j < length(t)$, denote the sequence composed of actions at positions $i$ to $j$ in log $t$. We also use $t_i$ as a shorthand for $t_{i..i}$. These may be easily defined as shown below.

**Definition 5.6.** The length and sublog functions are defined as follows:

$$length(\alpha_1\,; \ldots ; \alpha_n) \triangleq n \qquad (\alpha_1\,; \ldots ; \alpha_n)_{i..j} \triangleq \alpha_i\,; \ldots ; \alpha_j$$

With these definitions in place, we may now define the satisfaction relation. The rules for the constant $\mathbf{T}$, negation $\neg A$ and conjunction $A \wedge B$ are simply adaptations of the standard definitions of these to our setting. This is also the case for the existential quantifier $\exists x.A$. As we mentioned earlier, the bound variable $x$ is meant to range over channel names. This is reflected in the rule for $\exists x.A$, which states that this latter is satisfied by a log $t$ if there exists a channel name $n$ such that $t$ satisfies $A\{^n/_x\}$. The sometime modality $<\alpha>A$ is satisfied by a log $t$ if $\alpha$ is the $i$th action in $t$ and formula $A$ is satisfied by the sublog $t_{i+1..n}$ where $n$ is the length of log $t$. This does indeed correspond to the intuitive explanation we gave of the modality. A log $t$ is meant to be a complete account of the past actions of a system, ordered from the most recent action to the oldest. Hence, log $t_{i+1..n}$ denotes those actions that took place before action $\alpha$.

Figure 5.4 gives the full definition of satisfaction.

Figure 5.4. *The provenance logic: satisfaction*

| | | |
|---|---|---|
| (LSAT TRUE) | $t \models \mathbf{T}$ | |
| (LSAT NOT) | $t \models \neg A$ | $\triangleq t \not\models A$ |
| (LSAT AND) | $t \models A \wedge B$ | $\triangleq t \models A$ and $t \models B$ |
| (LSAT SOMETIME) | $t \models \mathord{<}\alpha\mathord{>}A$ | $\triangleq t_i = \alpha$ and $t_{i+1..n} \models A$ where $length(t) = n$ |
| (LSAT EXISTS) | $t \models \exists x.A$ | $\triangleq t \models A\{^n/_x\}$ for some $n \in C$ |

We wish to use formulae in the logic to describe and make statements about the past behaviour of a system. With this interpretation, the pre-order between formulae induced by logical implication $\Rightarrow$ corresponds quite naturally to the pre-order obtained when considering “how much information” a particular formula tells us about the past of a system. That is, given two formulae $A$ and $B$ both describing the past of the same system, one may ask: how much information does each formula tell us about the past of the system? Logical implication then answers this question in the relative sense where $A \Rightarrow B$ is interpreted as saying that $A$ tells us at least as much about the past as $B$, or in other words, that $A$ is at least as informative about the past as $B$.

We end this section by proving the following lemma, which we need in the proof of provenance correctness.

**Lemma 5.7.** *$t \models A$ and $A$ does not contain negation implies that $\alpha; t \models A$.*

*Proof.* We assume that $t \models A$ holds. We then proceed by induction on the last rule used in the derivation of this latter. In the case when this latter was derived using the rule for $\mathbf{T}$, $\alpha; t \models A$ holds vacuously. In the case for the sometime modality, we have that $A$ has the form $\mathord{<}\alpha'\mathord{>}A'$, $t = t'; \alpha'; t''$ and $t'' \models A'$. We find that two cases arise from this: the first is when $\alpha = \alpha'$ and the second is when $\alpha \neq \alpha'$. If $\alpha = \alpha'$, then by applying the rule for sometime we get $t'; \alpha'; t'' \models A'$, which follows from the induction hypothesis and the assumption $t'' \models A'$. If $\alpha \neq \alpha'$, then by applying the rule for sometime, we find that $\alpha$ is simply discarded with the other irrelevant parts of $t$ (that is $t'$) and we are left with $t'' \models A'$ which holds by assumption. The other cases are straightforward and may be shown to hold by simple application of the induction hypothesis. The condition that $A$ does not contain negation means that no subterm of $A$ (including $A$ itself) should be of the form $\neg B$ and is important as the statement of the lemma would not hold in the presence of negation. To see this, consider as a counter example the statement $\epsilon \models \neg(\mathord{<}\alpha\mathord{>}\mathbf{T})$ which is true since $\epsilon \not\models \mathord{<}\alpha\mathord{>}\mathbf{T}$. However, the statement $\alpha \models \neg(\mathord{<}\alpha\mathord{>}\mathbf{T})$ is false since $\alpha \models \mathord{<}\alpha\mathord{>}\mathbf{T}$ does hold. $\square$

## 5.3 Correctness of provenance

### 5.3.1 Denotation of provenance

We interpret the provenance $\kappa$ in an annotated value $p{:}\kappa$ as a set of assertions about the past of $p$. These assertions tell us about events that took place in the system and that are relevant to the value $p$. For example, consider the annotated value $p : a?\kappa \,; b!\kappa' \,; \kappa''$, its provenance tells us that (1) $p$ was most recently received by $a$ on a channel whose provenance was $\kappa$, (2) before that, it was sent by $b$ on a channel with provenance $\kappa'$, (3) and before that, it had provenance $\kappa''$, (4) $\kappa$ and $\kappa'$ in turn tell us about the past of the two channels used by $a$ and $b$, while $\kappa''$ tells us about the past of $p$ before it was sent by $b$ . It is important here to note that the provenance of $p$ does not reveal the identities of the channels used for communication, nor does it tell us about the ordering between events in $\kappa$ and those in $b!\kappa' \,; \kappa''$ or between events in $\kappa'$ and those in $\kappa''$.

Assertions such as those above may be encoded as formulae in the logic of Section 5.2. This is defined by the function $[\![-]\!] : \mathcal{V} \to \Phi$ which maps annotated values to logical formulae. In the definition of the function $[\![-]\!]$, we use a different grammar for provenance sequences than the one we originally gave in Figure 3.5. This is only done in order to simplify the definition of the denotation function. It is easy to check, however, that for our purposes the two grammars may be considered equivalent.

**Definition 5.8** (Denotation of provenance)**.** The function $[\![-]\!] : \mathcal{V} \to \Phi$ is defined inductively on the structure of provenance sequences as shown in Figure 5.5.

Figure 5.5. *Denotation of provenance sequences*

$$[\![p : \epsilon]\!] \triangleq \mathbf{T}$$
$$[\![p : a!\kappa \,; \kappa']\!] \triangleq \exists x.{<}a.\mathsf{snd}(x, p){>}([\![x : \kappa]\!] \wedge [\![p : \kappa']\!])$$
$$[\![p : a?\kappa \,; \kappa']\!] \triangleq \exists x.{<}a.\mathsf{rcv}(x, p){>}([\![x : \kappa]\!] \wedge [\![p : \kappa']\!])$$

The empty provenance sequence $\epsilon$ in the annotated value $p : \epsilon$ tells us nothing about the past of $p$, and hence $[\![p : \epsilon]\!]$ is taken to be the constant $\mathbf{T}$. The annotated values $p : a!\kappa \,; \kappa'$ and $p : a?\kappa \,; \kappa'$ denote that $p$ was sent (respectively received) sometime ago on some unknown channel $x$, and that the behaviour of the system up to that point may be described by $[\![x : \kappa]\!]$ and $[\![p : \kappa']\!]$. The conjunction here means that both formulae were true before the output (respectively input) action. However, the order between actions described by $[\![p : \kappa]\!]$ and $[\![x : \kappa']\!]$ is unknown as is the fact of whether these actions are distinct or not. What we mean by this last statement is that the unknown channel name $x$ may well be a copy of $p$ itself, in which case the two will probably share part of their past, and hence, the actions they describe would not be distinct. However, provenance alone does not tell us whether this is the case or not, and hence the same is reflected in logical formulae.

### 5.3.2 Monitored systems

In order to be able to assess the correctness of provenance, we introduce the notion of *monitored systems*. A monitored system is one where every action that takes place is recorded in a *global log*. The global log provides a repository where every action is logged and whose content is not accessible to principals and therefore can be more easily judged to be correct. Monitored systems are only meant to serve as *a proof tool*. They are aimed to help us prove the correctness of the distributed provenance tracking found in annotated values by relating it to a centralised global log. This global log is shared by the entire distributed system and takes the form of an ordered sequence of actions. This assumes that every two actions that take place within the system can be temporally ordered and therefore resembles an interleaving model of concurrency with a global clock. It should be noted however that this assumption does not affect the validity of the results proved in this section. They are assumptions on the proof tool and not on the actual calculus or any implementation of it. Proposition 5.10 shows that the semantics of monitored systems is faithful to that of the original calculus which ensures that any results we prove for monitored systems are indeed applicable to the original calculus. Intuitively, our chain of reasoning here can be understood as follows. We aim to prove that the provenance of every value in a system is correct. The provenance of each value can be seen as recording part of the history of the system. Therefore, the history of an annotated system as a whole is split between the different values it contains. Let us assume that there is a global log which records the history of the system in its entirety. Note that we do not need to worry that this assumption might be unfeasible in practice or wholly unrealistic. The reason is that we are only interested in using it as an aid in the proofs and therefore we only need to guarantee that it is faithful to the original semantics of the calculus. Now, assuming the existence of the global log, the provenance of a value is correct if what it tells us about the past of the system is corroborated by the global log. In other words, the provenance of a value is correct if its logical denotation according to the definitions of the previous section is true with respect to the log.

We use $M, N, \ldots$ to range over monitored systems and give their formal syntax below.

Figure 5.6. *Monitored systems: syntax*

| | |
|---|---|
| $M, N ::=$ | monitored systems |
| $t \triangleright S$ | system $S$ with global log $t$ |
| $(\nu n)M$ | restriction |
| $M \mid S$ | parallel composition |

The term $t \triangleright S$ denotes the monitored system composed of global log $t$ and system $S$. Restriction $(\nu n)M$ and parallel composition $M \mid S$ are needed to allow the global log to behave like other parts of the system with respect to scope extrusion and intrusion, giving it access to restricted or private channel names. More specifically, the form $(\nu n)M$ allows channel scopes to be extruded or extended to include the log whenever required, while the form $M \mid S$ is needed to allow channels whose scope includes the log but not some part of the system $S$. Note that, as expected, the above syntax permits exactly one global log per monitored system, allowing the log to be a record of everything that takes place within the system. We also define versions of structural congruence and reduction for monitored systems and denote them by $\equiv_m$ and $\rightarrow_m$ respectively. These are given in figures 5.7 and 5.8 respectively.

Figure 5.7. *Monitored systems: structural congruence*

| | |
|---|---|
| (MStr Alpha) | $M \equiv_\alpha M' \Rightarrow M \equiv_m M'$ |
| (MStr Log) | $S \equiv_a T \Rightarrow t \triangleright S \equiv_m t \triangleright T$ |
| (MStr Assoc) | $(t \triangleright S) \mid S' \equiv_m t \triangleright (S \mid S')$ |
| (MStr Res Res) | $(\nu n)(\nu m)M \equiv_m (\nu m)(\nu n)M$ |
| (MStr Res Par$_1$) | $(\nu n)M \mid S \equiv_m (\nu n)(M \mid S)$ if $x \notin fn(S)$ |
| (MStr Res Par$_2$) | $M \mid (\nu n)S \equiv_m (\nu n)(M \mid S)$ if $x \notin fn(M)$ |

Figure 5.8. *Monitored systems: reduction*

MRed Snd

$$\frac{}{t \triangleright a[m{:}\kappa_m \langle p{:}\kappa_p \rangle] \rightarrow_m a.\mathsf{snd}(m, p); t \triangleright m\langle\!\langle p : a!\kappa_m \,;\, \kappa_p \rangle\!\rangle}$$

MRed Rcv

$$\frac{j \in I \qquad \kappa_p \models \pi_j}{t \triangleright a[\Sigma_{i \in I} m{:}\kappa_m \, (\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p : \kappa_p \rangle\!\rangle \rightarrow_m a.\mathsf{rcv}(m, p); t \triangleright a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

MRed If$_t$

$$\frac{}{t \triangleright a[\mathsf{if}\ m{:}k_m = m{:}k'_m\ \mathsf{then}\ P\ \mathsf{else}\ Q\ ] \rightarrow_m a.\mathsf{ift}(m, m); t \triangleright a[P]}$$

MRed If$_f$

$$\frac{m \neq n}{t \triangleright a[\mathsf{if}\ m{:}k_m = n{:}k_n\ \mathsf{then}\ P\ \mathsf{else}\ Q\ ] \rightarrow_m a.\mathsf{iff}(m, n); t \triangleright a[Q]}$$

MRed Res

$$\frac{M \rightarrow_m M'}{(\nu n)M \rightarrow_m (\nu n)M'}$$

MRed Par

$$\frac{M \rightarrow_m M'}{M \mid S \rightarrow_m M' \mid S}$$

MRed Str

$$\frac{M \equiv_m N \qquad N \rightarrow_m N' \qquad N' \equiv_m M'}{M \rightarrow_m M'}$$

To help us in proving the correspondence between the semantics of monitored systems and the original provenance tracking semantics of the calculus, we define the *log erasure function* $|{-}|$. The log erasure function takes a monitored system and removes its global log, returning just the system part of it.

**Definition 5.9** (Log erasure function)**.** The function $|{-}|$ is defined inductively on the structure of monitored systems as follows:

$$|t \triangleright S| \triangleq S \qquad |(\nu n)M| \triangleq (\nu n)|M| \qquad |M \mid S| \triangleq |M| \mid S$$

It should be clear from the definition of $\rightarrow_m$ that the original provenance tracking semantics of systems is preserved. Indeed, the only difference between the reduction relation of monitored systems $\rightarrow_m$ and that of annotated systems $\rightarrow_a$ is the addition of the global log in the case of monitored systems. Every transition from a monitored system $M$ to a monitored system $M'$ using the relation $\rightarrow_m$ has a corresponding one from the system $|M|$ to the system $|M'|$ using the relation $\rightarrow_a$. Note that $|M|$ and $|M'|$ represent the results of removing the global logs from systems $M$ and $M'$ respectively. This implies that the semantics of monitored systems *does not add* any extra behaviours to systems. Similarly, it *does not remove* any behaviours from systems. That is, every transition from a system $|M|$ to a system $|M'|$ using the relation $\rightarrow_a$ has a corresponding one from system $M$ to system $M'$ using the relation $\rightarrow_m$. This is formalised and proved in Proposition 5.10. This proposition is very important as it guarantees that any results we prove for monitored systems are indeed applicable to the original calculus.

**Proposition 5.10.** *$M \rightarrow_m M'$ implies that $|M| \rightarrow_a |M'|$. Vice versa, $|M| \rightarrow_a S$ implies that $M \rightarrow_m M'$ for some $M'$ such that $|M'| = S$.*

*Proof.* The proof for this is by simple rule induction. We assume that $M \rightarrow_m M'$ and prove that $|M| \rightarrow_a |M'|$ follows from it. In each case, there is a corresponding rule to use to derive $\rightarrow_a$ transitions by simply dropping the logs. The other direction follows in a similar manner where the transitions of the relation $\rightarrow_m$ lead to monitored systems that are the same as those found in the relation $\rightarrow_a$ but with the addition of a log. Erasing this shows that the systems obtained are indeed the same. □

### 5.3.3 Correctness

Now we look at the correctness property of provenance tracking. A provenance sequence $\kappa$ in an annotated value $p : \kappa$ is considered correct if what it tells us about the past of $p$ agrees with what actually took place. This is defined relative to the global log which is assumed to be a correct and complete record of the past of a system. If every value in a system has correct provenance, then we say that the system as a whole has correct provenance. Theorem 5.13 states that provenance correctness is preserved by the reduction relation. That is, starting from a system with correct provenance, we are guaranteed to get a system with correct provenance after reduction. Definition 5.11 makes use of two auxiliary functions: $log(-)$, which returns the global log of a monitored system, and $values(-)$, which returns the set of annotated values of a monitored system. The definition of $log(-)$, by induction on the structure of monitored systems, is straightforward and simply returns the single term of a monitored system that has the form $t$. This is mostly the case for $values(-)$ too and hence, we only discuss the most interesting cases below. Note that in the rest of this section, we use the meta variables $V, U$ to range over the set of plain values extended with the symbol $\star$. The special symbol $\star$ is used to denote a restricted channel name that is not known to the global log. The set of values in a monitored system is defined to be that in its system part (i.e. we ignore the global log and top level restrictions). This is expressed by the following three rules:

$$values(t \triangleright S) \triangleq values(S) \qquad values((\nu n)M) \triangleq values(M)$$
$$values(M \mid S) \triangleq values(M) \cup values(S)$$

For systems, we proceed simply by gathering annotated values, that is subterms of the form $v : \kappa$, and substituting $\star$ for any restricted channel names. So, for example, we have that:

$$values(a[P]) \triangleq values(P) \qquad values(m\langle\!\langle v : \kappa \rangle\!\rangle) \triangleq \{v : \kappa\}$$
$$values(S \mid S') \triangleq values(S) \cup values(S') \qquad values((\nu n)S) \triangleq values(S)\{^{\star}/_{n}\}$$

Note that restriction here is treated differently from that at the top level of monitored systems. The rationale behind this discrepancy is that restricted names at the top level are known to the global log whereas those occurring here are not. The substitution is done to avoid any clashes with names appearing in the log and is inspired by Lhoussaine and Sassone [45]. Definition of $values(P)$ for processes $P$ is similar. The complete definitions of $log(-)$ and $values(-)$ are given below.

Figure 5.9. *Definition of $log(-)$ and $values(-)$*

DEFINITION OF $log(-)$.

$log(t \triangleright S) = t$ $\quad$ $log((\nu n)M) = log(M)$ $\quad$ $log(M \mid S) = log(M)$

DEFINITION OF $values(-)$.

$values(t \triangleright S) = values(S)$ $\quad$ $values(M \mid S) = values(M) \cup values(S)$

$values((\nu n)M) = values(M)$

$values(a[P]) = values(P)$ $\quad$ $values(m\langle\!\langle w \rangle\!\rangle) = values(w)$

$values((\nu n)S) = values(S)\{^\star/_m\}$ $\quad$ $values(S \mid S') = values(S) \cup values(S')$

$values(w\,\langle w' \rangle) = values(w) \cup values(w')$ $\quad$ $values(* P) = values(P)$

$values((\nu n)P) = values(P)\{^\star/_m\}$ $\quad$ $values(P \mid P') = values(P) \cup values(P')$

$values(\Sigma_{i \in I} w\,(x, \pi_i).P_i) = \bigcup_{i \in I} values(P_i) \cup values(w)$

$values(\text{if } w = w' \text{ then } P \text{ else } Q\,) = values(w) \cup values(w') \cup values(P) \cup values(Q)$

$values(v : \kappa) = \{v : \kappa\}$ $\quad$ $values(x) = \emptyset$

**Definition 5.11.** A monitored system $M$ *has correct provenance* if for all $V : \kappa$ in $values(M)$, we have that $log(M) \models [\![V : \kappa]\!]$.

**Lemma 5.12.** *$M$ has correct provenance and $M \equiv_m M'$ implies that $M'$ has correct provenance.*

*Proof.* We assume that $M$ has correct provenance and that $M \equiv_m M'$. To prove that $M'$ has correct provenance, we proceed by induction on the derivation of $M \equiv_m M'$. The details of the proof are straightforward but tedious and are hence omitted. □

**Theorem 5.13.** *(Provenance correctness). $M$ has correct provenance and $M \rightarrow_m M'$ implies that $M'$ has correct provenance.*

*Proof.* We assume that $M$ has correct provenance and that $M \rightarrow_m M'$. We show that $M'$ has correct provenance by induction on the derivation of $M \rightarrow_m M'$. Below, we illustrate the cases when this latter was derived using the rules MRed Snd, MRed Rcv and MRed Res. Other cases proceed in a similar manner.

Case MRed Snd. This means that $M$ was of the form $t \triangleright a[m{:}\kappa_m \langle p{:}\kappa_p\rangle]$ and $M'$ was of the form $a.\mathsf{snd}(m,p); t \triangleright m\langle\!\langle p{:}a!\kappa_m;\kappa_p\rangle\!\rangle$. Since $M$ has correct provenance, then we know that $t \models [\![m:\kappa_m]\!]$ and $t \models [\![p:\kappa_p]\!]$. To prove that $M'$ has correct provenance, we need to show that $a.\mathsf{snd}(m,p); t \models [\![p: a!\kappa_m\,;\kappa_p]\!]$, that is we need to show that $a.\mathsf{snd}(m,p); t \models \exists x.{<}a.\mathsf{snd}(x,p){>}([\![p{:}\kappa_p]\!] \wedge [\![x{:}\kappa_m]\!])$. From this latter and by application of the rules LSat Exists (substituting $m$ for $x$) and LSat Sometime, we get $t \models [\![p{:}\kappa_p]\!] \wedge [\![m{:}\kappa_m]\!]$. This then is concluded to hold by using the rule LSat And and the assumptions $t \models [\![m:\kappa_m]\!]$ and $t \models [\![p:\kappa_p]\!]$. Hence, $M'$ has correct provenance.

Case MRed Rcv. This means that $M$ was of the form $t \triangleright a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \;\mathsf{as}\; x).P_i] \mid m\langle\!\langle p: \kappa_p\rangle\!\rangle$ and $M'$ was of the form $a.\mathsf{rcv}(m,p); t \triangleright a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]$. Since $M$ has correct provenance, then we know that $t \models [\![m{:}\kappa_m]\!]$, $t \models [\![p{:}\kappa_p]\!]$ and for all $V{:}\kappa$ in $values(P_i)$ (for all $i \in I$), $t \models [\![V:\kappa]\!]$. To prove that $M'$ has correct provenance, we need to show that for all $V{:}\kappa$ in $values(P_j\{^{p:a?\kappa_m;\kappa_p}/_x\})$, it is the case that $a.\mathsf{rcv}(m,p); t \models [\![V{:}\kappa]\!]$ holds. It can be shown that $values(P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}) = values(P_j)$ if $x \notin fv(P_j)$ and $values(P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}) = values(P_j) \cup \{p : a?\kappa_m\,;\kappa_p\}$ if $x \in fv(P_j)$. We know that for all $V:\kappa$ in $values(P_j)$, $t \models [\![V:\kappa]\!]$ holds, and hence by application of Lemma 5.7, we are able to infer that $a.\mathsf{rcv}(m,p); t \models [\![V:\kappa]\!]$ holds too. This means that the first case (i.e., when $x \notin fv(P_j)$) is satisfied. For the second case, we still need to show that $a.\mathsf{rcv}(m,p); t \models [\![p: a?\kappa_m\,;\kappa_p]\!]$. This latter simplifies to $a.\mathsf{rcv}(m,p); t \models \exists x.{<}a.\mathsf{rcv}(x,p){>}([\![p:\kappa_p]\!] \wedge [\![x:\kappa_m]\!])$. Using the rules LSat Exists and LSat Sometime, we get $t \models [\![p:\kappa_p]\!] \wedge [\![m:\kappa_m]\!]$. This latter then follows from the assumptions $t \models [\![m:\kappa_m]\!]$ and $t \models [\![p:\kappa_p]\!]$ by application of the rule LSat And. Hence, $M'$ has correct provenance.

Case MRed Res. This means that $M \rightarrow_m M'$ was derived as follows:

$$\frac{N \rightarrow_m N'}{(\nu n)N \rightarrow_m (\nu n)N'}$$

where $M$ is of the form $(\nu n)N$ and $M'$ is of the form $(\nu n)N'$. Since $(\nu n)N$ has correct provenance, then so does $N$ (this follows from the fact that $values((\nu n)N) = values(N)$ and $log((\nu n)N) = log(N)$). This implies, by the induction hypothesis, that $N'$ has correct provenance. From this latter, it follows that $(\nu n)N'$ has correct provenance. Hence, $M'$ has correct provenance. □

## 5.4 Concluding remarks

In this chapter, we looked at the semantics of provenance. We started by observing that not all terms admitted by the grammar of provenance sequences are semantically meaningful. The intuition behind this was quite simple; a provenance sequence is semantically meaningful only if it could arise from a valid sequence of transitions according to the semantics of the calculus. We formalised this in the definition of well-formed provenance sequences and gave a sound and complete syntactic characterisation of them.

We noted that well-formedness is a weak property however. The only guarantee it gives us is that the provenance could arise in a valid sequence of transitions. However, it is entirely possible that this sequence of transitions, while valid, is only hypothetical. That is, it never actually took place. This led us to define a stronger properly, correctness. In order to define correctness, we introduced a temporal logic with a past modality that enabled us to make statements about the past behaviour of systems. With this logic, we were able to give a denotational semantics of provenance sequences in the form of formulae in the logic. The formal definition of correctness then was simple; a provenance sequence is correct if its denotation, that is the logical statement it makes about the past behaviour of the system, is true. To capture the past behaviour of systems in a simple form, we defined monitored systems. These are systems that are augmented with a global log of all past system behaviour. The truth or falsehood of what provenance tells us about the past is defined with respect to this global log. Finally, we proved that the provenance tracking reduction relation of the calculus always preserves the correctness of provenance.

# Chapter 6

# Provenance as Types

The calculus we defined and explored in the previous chapters relies on *run time* provenance tracking and checking. That is, updating the provenance annotations of values and verifying them against the provenance policies of principals are both done dynamically as the system evolves. This results in performance overhead at run time and motivates the need for a static approach to provenance tracking and checking. Such an approach would consist of analysing systems at compile time and *conservatively approximating* the provenance of their values in order to guarantee that no provenance policies would ever be violated at run time. For systems that *pass* the static analysis, this would mean that dynamic provenance tracking and checking become redundant and therefore they may be dropped. It is the aim of this chapter to propose and study such a static analysis.

To illustrate how the proposed static analysis works, consider the system $S$ defined below:

$$S \triangleq a[m\,\langle p\rangle \mid n\,\langle q\rangle] \mid b[m\,(a!\mathsf{Any}\ \mathsf{as}\ x).n\,\langle x\rangle] \mid c[n\,(\mathsf{Any}\ ;\ a!\mathsf{Any}\ \mathsf{as}\ x).P]$$

Note that this system is an annotated one; we are simply using our convention of dropping all $\epsilon$ provenance sequences from values. The aim of the static analysis is to avoid the need to:

- annotate values with provenance sequences,
- update these annotations every time the system makes a computational step, and
- check the provenance policies of principals against these annotations before data can be consumed.

To achieve these aims, the static analysis we propose considers all the policies in the system that need to be satisfied, which in the case of the system above would be the two patterns $a!\mathsf{Any}$ and $\mathsf{Any} \,;\, a!\mathsf{Any}$ imposed on channels $m$ and $n$ respectively. For this system to pass the static analysis, these two patterns would have to be satisfied in *every* run of the system. Naturally, this could be achieved by simply enumerating all the possible runs of the system above and verifying that indeed, the two patterns would be satisfied in every run. For a system as simple as the one above, this is actually not very difficult. However, such a naive approach would quickly become intractable as the system gets larger and more complex. The static analysis we propose in this chapter on the other hand aims to be much more efficient. It proceeds by treating the two aforementioned patterns as channel constraints that all values sent on those channels need to satisfy. This means that all values sent on channel $m$ need to satisfy the pattern $a!\mathsf{Any}$ while all values sent on channel $n$ need to satisfy the pattern $\mathsf{Any} \,;\, a!\mathsf{Any}$. For channel $m$, there is only one output we need to consider which is $m\,\langle p\rangle$ at $a$. This output would result in value $p$ with provenance $a!\epsilon$, and since this satisfies the pattern $a!\mathsf{Any}$, we can conclude that the constraint of channel $m$ is satisfied. For channel $n$, we have two outputs, $n\,\langle q\rangle$ at $a$ and $n\,\langle x\rangle$ at $b$. We can use similar reasoning to that of channel $m$ to deduce that $n\,\langle q\rangle$ would satisfy the constraint $\mathsf{Any} \,;\, a!\mathsf{Any}$ on channel $n$. The same is not possible with $n\,\langle x\rangle$ however as we need to know the provenance of variable $x$ first. This depends on what value principal $b$ ends up receiving on channel $m$ and substituting for variable $x$, and in general we would not have that information at compile time. What we have instead is the knowledge that whatever value $b$ receives, its provenance has to satisfy the constraint $a!\mathsf{Any}$ that $b$ is imposing on channel $m$. Therefore, we can safely use $a!\mathsf{Any}$ as an approximation of the provenance of variable $x$ (or more correctly, of the value that would ultimately be substituted for variable $x$). We can show that any provenance sequence that satisfies $a!\mathsf{Any}$ must also satisfy $\mathsf{Any} \,;\, a!\mathsf{Any}$. Therefore, we can conclude that all values sent on $n$ satisfy the constraint imposed on it. With this, we can declare that system $S$ passes the static analysis and hence, there is no need to perform the provenance checks at run time, or for that matter, to keep track of the provenance of values. It is important here to note the distinction between the *actual* provenance of data and the annotations associated with it. In the calculus studied so far, the provenance annotations are meant as a *concrete record* of the provenance of data, and hence the policies of principals could be verified against these annotations. With static analysis, there is no need anymore to keep and update these provenance annotations as the provenance policies are verified in a different way. In other words, with run time provenance tracking and checking, the provenance of data is explicitly represented by the provenance annotations of values; while when using the static analysis, no such explicit representation is needed and provenance is kept implicit.

The structure of this chapter is as follows. Section 6.1 gives an overview of the static analysis with the aim of making it easier to navigate the following sections. In particular, it provides a summary of the three stages in which we introduce the static analysis. The first stage, which is covered in Section 6.2, starts with a restricted version of the calculus without the choice operator and proceeds to eliminate provenance checking from the calculus. However, at this stage, we keep the tracking of provenance in order to allow us to carry out the necessary proofs. We prove that the static analysis we propose guarantees that even without provenance checking, systems that pass the analysis never violate the provenance policies of their principals at run time. In Section 6.3, we look at the second stage where provenance tracking too is eliminated from the restricted calculus, and this without impacting the results of the first stage. Finally, the third stage is to adapt the static analysis to the full calculus; we look at this in Section 6.4. Section 6.5 concludes the chapter.

## 6.1 Overview

The static analysis we propose in this chapter takes the form of a type and effect system. We treat the provenance annotations of values as types and the provenance checks in inputs as run time type checks. Provenance as types provides for a very interesting topic of study in its own right since it corresponds to dynamic, behavioural types as we will see later in the chapter. For instance, consider principal $b$ from the example above which when written without dropping the $\epsilon$ provenance annotations would be $b[m{:}\epsilon\,(a!\mathsf{Any}\ \mathsf{as}\ x).n{:}\epsilon\,\langle x\rangle]$. In our type system, values $m$ and $n$ would both have the type $\epsilon$ while the variable $x$ would be assigned the type $a!\mathsf{Any}$. The type system then proceeds to approximate how these types would evolve in different runs of the system. The approximated types (or provenance) of output values on a particular channel are then verified against the patterns associated with inputs on that channel. If all of these pass the verification, then the system is declared well-typed. A system that is well-typed with respect to the type and effect system of this chapter is guaranteed to always respect the provenance policies of principals.

Dealing with input-guarded choice would overcomplicate the type system. The reason is that from a typing point of view, the variable $x$ in the process $\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i\ \mathsf{as}\ x).P_i$ has the union type $\bigcup_{i\in I}\pi_i$. This in itself is not difficult to handle in the type system. However, the semantics of input-guarded choice in the current version of the calculus dictates that each $\pi_i$ should trigger the corresponding continuation $P_i$. This would not be possible without dynamic type checking. Therefore, we opt to focus our attention in the next two sections on the subset of the calculus where the guarded choice construct, $\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i\ \mathsf{as}\ x).P_i$, is restricted to empty and singleton indexing sets $I$. Alternatively, this may be viewed as simply replacing $\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i\ \mathsf{as}\ x).P_i$ with nil 0 and input prefix $m{:}\kappa_m\,(\pi\ \mathsf{as}\ x).P$, and for simplicity, we shall use these latter as notation and drop the choice operator and the subscripts for the remainder of this chapter. This means that for the purposes of the next two sections, we should consider the rule ARed Rcv to be defined as:

$$\frac{\kappa_p \models \pi}{a[m{:}\kappa_m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:a?k_m;\kappa_p}/_x\}]} \tag{6.1}$$

Nevertheless, in Section 6.4, we shall discuss ways in which the type system could be extended to the full calculus.

The aim of the type system is to eliminate both dynamic provenance tracking and checking. What is meant by this is that, with the guarantees offered by the type system, the main rules of the calculus, i.e., the rules for input and output could be replaced with the following:

$$\frac{}{a[m\,\langle p\rangle] \rightarrow_a m\langle\!\langle p\rangle\!\rangle} \qquad \frac{}{a[m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p\rangle\!\rangle \rightarrow_a a[P\{^{p}/_x\}]} \tag{6.2}$$

These two rules look almost the same as their counterparts in the plain calculus. They only differ in that the input construct still has the pattern $\pi$. However, this latter does not have any impact on the dynamic semantics of the rule itself. The provenance policy expressed by this pattern would instead be verified statically by the type system. Getting rid of both provenance tracking and checking at the same time makes the proofs of type preservation more complicated however. For this reason, we do this in two steps. Firstly, we get rid of provenance checking. This means that we first drop the premise $\kappa_p \models \pi$ from the receive rule, replacing the definition:

$$\frac{\kappa_p \models \pi}{a[m{:}\kappa_m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:a?k_m;\kappa_p}/_x\}]} \tag{6.3}$$

with

$$\frac{}{a[m{:}\kappa_m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:a?k_m;\kappa_p}/_x\}]} \tag{6.4}$$

We prove that, in well-typed systems, principals are still guaranteed to only consume data with the right provenance. After that, we prove that without provenance checking, provenance tracking becomes redundant and can be dropped too. This allows us to replace the input and output rules:

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p{:}a!\kappa_m\,;\kappa_p\rangle\!\rangle} \qquad \frac{}{a[m{:}\kappa_m\,(\pi \text{ as } x).P] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}]} \tag{6.5}$$

with:

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p{:}\kappa_p\rangle\!\rangle} \qquad \frac{}{a[m{:}\kappa_m\,(\pi \text{ as } x).P] \mid m\langle\!\langle p:\kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:\kappa_p}/_x\}]} \tag{6.6}$$

Note the difference between the two sets of rules. The first two rules lack provenance checking in the input rule but do perform provenance tracking as before. On the other hand, the second two do not perform provenance tracking. Instead, the provenance annotations of values remain the same after the transitions. Since we assume that all systems start in an initial state, i.e., without any provenance, we would expect the provenance sequences in these two rules to be simply $\epsilon$. Dropping these as per our convention gives us the rules of 6.2.

## 6.2 Eliminating provenance checking

For the purposes of this section, we introduce a statically typed version of the provenance tracking reduction relation, which we denote by $\rightarrow_s$. The relation $\rightarrow_s$ does not include dynamic provenance checks as it is meant to rely on static type checking. As a result, the definition of $\rightarrow_s$ only differs from that of $\rightarrow_a$ in the rule for input, which is given as follows:

$$\frac{}{a[m{:}\kappa_m\,(\pi \text{ as } x).P] \mid m\langle\!\langle p:\kappa_p\rangle\!\rangle \rightarrow_s a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

As can be seen from the rule above, the only difference with respect to the rule ARed Rcv is the omission of the premise $\kappa_p \models \pi$. The static analysis ensures that in well-typed systems, principals always receive data with provenance that meets their requirements, and hence the dynamic check is not required. The other rules of $\rightarrow_s$ are the same as those of the relation $\rightarrow_a$, and we use the convention of naming them similarly but with the initial "A" replaced with an "S". So for example, the statically checked counterparts of the rules ARed Snd, ARed Rcv and ARed Par are SRed Snd, SRed Rcv and SRed Par respectively.

### 6.2.1 A type and effect system for provenance

The idea behind the type and effect system proposed in this chapter is to view each channel name as having a set of constraints, specified by the consumers of data on that channel, and required of producers of data on the channel. The constraints of a channel relate to the provenance of data that it may *safely* carry. Every use of the channel for input imposes a new constraint; the pattern that the principal specifies for data it considers to have acceptable provenance. Every use of the channel for output produces data that is required to satisfy the constraints of the channel; the provenance of this data must satisfy every pattern in the channel's set of constraints. More specifically, every process of the form $w\,(\pi \text{ as } x).P$ is viewed as imposing a constraint. This constraint consists of the pattern $\pi$ which the provenance of every value that may be received by this process needs to satisfy. In the original rule for input, the premise $\kappa_p \models \pi$ represents a check that is performed at run time to ensure the provenance of the value satisfies the pattern. The value is only consumed if this check passes. Values whose provenance does not meet any of the constraints of the channel would never be consumed by any principal. Hence, their existence in the system would not pose any issues with respect to the provenance policies of principals. Contrast this with what the type and effect system does. Since the run time provenance check is not present in the calculus anymore, the type system needs to ensure that the provenance of *every* value that could potentially be sent on a particular channel satisfies *all* the constraints of the channel. The provenance of data is not constant however; it changes as the system evolves. The type system attempts to predict how the provenance would change at run time in order to ensure that constraints are satisfied against the up-to-date provenance. Naturally, this prediction is only an approximation of the actual provenance that the data would have at run time. To ensure safety, this approximation is conservative, opting to reject possibly safe systems rather than risk admitting unsafe ones.

#### 6.2.1.1 Types and pattern satisfaction

**Types.** The type system treats provenance sequences, patterns, as well as any combination of these two as types. The provenance sequence $\kappa_p$ in the annotated value $p : \kappa_p$ is considered as the type of value $p$. The pattern $\pi$ in the input construct $w\,(\pi \text{ as } x).P$ is considered as the type of variable $x$. Combinations of provenance sequences and patterns are needed to approximate how the provenance of values is likely to evolve at run time. For instance, consider the principal $a[m{:}\kappa_m\,(\pi \text{ as } x).n{:}\kappa_n\,\langle x\rangle]$, which listens on channel $m$ for a value whose provenance must satisfy the pattern $\pi$ and then forwards this value onto channel $n$. Assuming that $a$ is able to find a value that satisfies the pattern $\pi$, say $p : \kappa_p$, then we would expect this value to be substituted for $x$ and sent on channel $n$. This sequence of transitions would result in the message $n\langle\!\langle p : a!\kappa_n \,;\, a?\kappa_m \,;\, \kappa_p\rangle\!\rangle$. However, principal $a$ could also receive a different value with a different provenance sequence. In fact, it could receive any value with any provenance as long as this latter satisfies the pattern $\pi$. We would not in general know what value is received by $a$ or what provenance it has until run time. All we know at compile time is that the provenance of this value must satisfy $\pi$. Therefore, we can conclude that the value that is going to be substituted for variable $x$ at run time will have provenance that looks like $p : a!\kappa_n \,;\, a?\kappa_m \,;\, \pi$. Note the pattern $\pi$ at the end of this provenance. The type and effect system uses types such as $a!\kappa_n \,;\, a?\kappa_m \,;\, \pi$ to approximate the provenance of variables at run time.

We use $\mathcal{T}$ to denote the set of types and use $t, t', \ldots$ to range over its elements. The syntax of types is given in Figure 6.1. It is easy to check that this syntax includes both provenance sequences and patterns as subsets, and that it is sufficient to account for the approximation of any run time provenance such as the one from the previous example.

Figure 6.1. *Syntax of types*

| | |
|---|---|
| $\mathcal{T}$ ::= | provenance types |
| $\epsilon$ | empty environment |
| $a!t$ | send event |
| $a?t$ | receive event |
| $t \,;\, t'$ | concatenation |
| $\pi$ | pattern |

**Satisfaction.** Consider again the example system $a[m{:}\kappa_m\,(\pi \text{ as } x).n{:}\kappa_n\,\langle x\rangle]$. In our previous discussion, we said that the type system would assign the type $a!\kappa_n \,;\, a?\kappa_m \,;\, \pi$ to variable $x$ after the output action. For this example to be well-typed, the type $a!\kappa_n;a?\kappa_m;\pi$ needs to satisfy all the constraints of channel $n$. If we take the system in isolation, then there are no input actions involving channel $n$ and therefore no constraints on it. Hence, the example is trivially well-typed. However, what if we were to compose this system in parallel with another one, say $b[n{:}\epsilon\,(\pi' \text{ as } y).P]$ for instance. This would mean that $n$ would now have the constraint $\pi'$ and therefore $a!\kappa_n \,;\, a?\kappa_m \,;\, \pi$ would be required to satisfy it. Satisfaction of patterns has only been defined with respect to provenance sequences though, and therefore, we have to define what it means for a type to satisfy a pattern. This will depend on the specifics of the pattern language used in the calculus. For the purposes of this chapter, we assume that the pattern language is equipped with a satisfaction relation that is defined for both provenance sequences and types. Technically, the satisfaction relation need only be defined for types since these include provenance sequences as a subset. Definition 6.1 gives the new specification of pattern languages.

**Definition 6.1.** A pattern matching language is a pair $(\Pi, \models)$ where $\Pi$ is a set of patterns, ranged over by $\pi, \pi', \ldots$, and $\models \subseteq \mathcal{T} \times \Pi$ is the pattern satisfaction (or matching) relation, a relation between provenance types and patterns.

In addition to single pattern satisfaction $t \models \pi$, the type system also makes use of a generalised notion of satisfaction of the form $t \models \{\pi_1, \ldots, \pi_n\}$. This is defined below and can be simply taken as syntactic sugar for $\bigwedge_{i \in 1..n} t \models \pi_i$.

**Definition 6.2.** We say that a type $t$ satisfies patterns $\pi_1, \ldots, \pi_n$, written $t \models \{\pi_1, \ldots, \pi_n\}$, when it is the case that $t \models \pi_i$ for all $i \in 1..n$.

**Subtyping.** We assume the existence of a subtyping relation $\leq$ between types. In particular, we assume that $\leq$ satisfies two properties:

- Subsumption: if $t \leq t'$ and $t' \models \pi$ then $t \models \pi$.
- Compositionality: if $t \leq t'$ then:
  - $a!t \leq a!t'$ for all principal names $a$.
  - $a?t \leq a?t'$ for all principal names $a$.
  - $t\,;t'' \leq t'\,;t''$ for all types $t''$.

Subtyping will be useful when formulating the Substitution Lemma as it allows us to substitute a value of type $t$ with another of type $t'$ as long as $t'$ is a subtype of $t$, all the while ensuring that the satisfaction of any constraints remains intact.

#### 6.2.1.2 Type and constraint environments

**Type environments.** A type system needs to be compositional. That is, it needs to allow us to type check the different parts of a given system separately and then put them together, obtaining a type derivation for the entire system. To accomplish this, type systems introduce techniques for representing the interfaces of systems that are relevant for type checking. That way, when composing multiple systems together, only their interfaces need to be checked to decide whether the compound system is well-typed or not. Put another way, a type system would use an abstraction of the context in which a system is to be run in order to type check it. If the system passes the type checking, then it may be safely plugged into any environment that is consistent with the interface provided by the abstraction. The simplest kind of these interfaces is probably the one given by *type environments*, functions that map free variables to types. Type environments allow us to determine the acceptable types that may be assigned to the free variables of a system and still retain its well-typedness.

We use the letters $\Delta, \Gamma, \ldots$ to range over type environments. We denote the assignment of type $t$ to variable $x$ by the standard notation $x : t$, and use $\Delta\,,\Gamma$ for the domain disjoint union of type environments $\Delta$ and $\Gamma$. If $x$ is in the domain of type environment $\Delta$, then we use $\Delta(x)$ to denote the type assigned to it. That is, $\Delta(x) = t$ when it is the case that $x : t \in \Delta$. It is worth noting here that there is no need for type environments to keep track of the types of free names. The reason is that free names, like all other values, are annotated with their provenance. The syntax of type environments is summarised in Figure 6.2.

Figure 6.2. *Syntax of type environments*

| | |
|---|---|
| $\Gamma ::=$ | typing environments |
| $\varnothing$ | empty environment |
| $x : \pi$ | type assignment |
| $\Delta\,,\Gamma$ | concatenation |

For the purposes of type checking, we extend the annotation extraction function $\|-\|$ of Definition 4.4 to variables. This is done by defining the function $\|-\|$ with respect to an environment $\Gamma$. We denote the generalised function by $\|-\|_\Gamma$ and give its definition below.

**Definition 6.3** (Extended annotation extraction function)**.** The annotation extraction function is extended to variables as follows:

$$\|p : \kappa\|_\Gamma \triangleq \kappa \qquad \|x\|_\Gamma = \Gamma(x)$$

**Constraint environments.** In addition to type environments, the type and effect system we propose also introduces the notion of *constraint environments*. As its name implies, a constraint environment acts as an abstraction of the constraints of the context and therefore is meant to account for all the input patterns of the context. If a system is well-typed with respect to a particular constraint environment, then it is guaranteed to only produce data with provenance that satisfies the constraints specified by the constraint environment. Therefore, we may safely plug it into any context that is consistent with this constraint environment, certain that it would never violate the provenance policies of principals in this context.

We need a way for constraint environments to refer to the bound names and bound variables of systems from outside their scope; we do this by using the notion of *abstract names* and *abstract variables* [45]. Abstract names and abstract variables can be thought of simply as references to bound names and bound variables that should be kept globally unique, in order to make it easier for the type system to compose multiple subsystems together when type checking. They are purely a technical tool used for the purposes of the type system. We use $\mathsf{m}, \mathsf{n}, \ldots$ to range over abstract names, and $\mathsf{x}, \mathsf{y}, \ldots$ to range over abstract variables. We let the meta-variables $V, U, \ldots$ range over plain values, variables, abstract names and abstract variables. Constraint environments contain the constraints that principals impose on channels. These as we explained are the patterns that specify what provenance is considered acceptable for those channels, and are represented in constraint environments by mappings of the form $V : \mathsf{c}(\pi)$. This latter denotes that $V$ has the constraint $\pi$, i.e., channel $V$ is required to input data with provenance that matches the pattern $\pi$. As the calculus allows principals to bind channels to variables at run time, constraint environments also need a way to keep track of such bindings. We do this by adding two more types of mappings to constraint environments. The first, denoted by $V\langle V'\rangle$ and referred to as an output link, means that $V'$ may be sent on $V$; while the second, denoted by $V(V')$ and referred to as an input link, means that $V'$ may be received on $V$. To understand why these two types of mappings are needed, consider the example system $S$ defined as follows:

$$S \triangleq a[m\,\langle n\rangle \mid n\,\langle p\rangle] \mid b[m\,(a!\mathsf{Any}\ \mathsf{as}\ x).x\,(\mathsf{Any}\,;c!\mathsf{Any}\ \mathsf{as}\ y)]$$

To decide whether $S$ is well-typed or not, we start by extracting all the constraints that it contains to build its constraint environment. System $S$ contains two inputs, leading to the two constraints $m : \mathsf{c}(a!\mathsf{Any})$ and $x : \mathsf{c}(\mathsf{Any}\,; c!\mathsf{Any})$. For the first constraint, $m : \mathsf{c}(a!\mathsf{Any})$, we look for all the outputs on channel $m$ and we find only one, $m\,\langle n\rangle$ at principal $a$. This would lead to message $m\langle\!\langle n : a!\epsilon\rangle\!\rangle$, and checking the provenance of its value we find that it satisfies the pattern $a!\mathsf{Any}$. Since there are no more outputs on channel $m$, we can conclude that its constraints are satisfied. Now we move on to the second constraint $x : \mathsf{c}(\mathsf{Any}\,; c!\mathsf{Any})$. We find that this constraint involves the variable $x$ and so in order to guarantee that it would be satisfied at run time, we need to know what channel names, if any, may be substituted for variable $x$. To be able to do that, we have to analyse the communication structure of the system. The results of this analysis are represented in constraint environments by output and input links. The output and input links are derived as follows. Working backwards from the input $x\,(\mathsf{Any}\,;\,c!\mathsf{Any}\ \mathsf{as}\ y)$, we find that $x$ is bound to the value received on channel $m$ by the input $m\,(a!\mathsf{Any}\ \mathsf{as}\ x)$. This fact is represented by the input link $m(x)$. Now we consider what values may be received on $m$, which we do by looking at all the outputs on channel $m$. We find one such output, $m\,\langle n\rangle$ at principal $a$, giving rise to the output link $m\langle n\rangle$. Together, the two links $m\langle n\rangle$ and $m(x)$ tell us that variable $x$ may be substituted with channel name $n$ at run time. This means that any output on channel $n$ would have to satisfy the constraints of variable $x$. The only output on channel $n$ is $n\,\langle p\rangle$ found at principal $a$. This output if executed would lead to the message $n\langle\!\langle p : a!\epsilon\rangle\!\rangle$, which means the provenance sequence $a!\epsilon$ has to satisfy the pattern $\mathsf{Any}\,;\,c!\mathsf{Any}$. Since this is not the case, we have to conclude that the constraint $x : \mathsf{c}(\mathsf{Any}\,;\,c!\mathsf{Any})$ is not satisfied, and hence, the system above is not well-typed.

We use the letters $\Theta$, $\Xi$, ... to range over constraint environments. We denote by $\varnothing$ the empty constraint environment and by $\Theta\,,\,\Xi$ the union of constraint environments $\Theta$ and $\Xi$. Note that $\Theta$ and $\Xi$ are not required to be domain disjoint as the same channel name may have multiple constraints for example. We have already seen the three types of mappings that a constraint environment may contain; channel constraint $V : \mathsf{c}(\pi)$, output link $V\langle V\rangle$, and input link $V(V)$. The syntax of constraint environments is summarised in Figure 6.3.

Figure 6.3. *Syntax of constraint environments*

| | | | |
|---|---|---|---|
| $\Theta ::=$ | constraint environments | $C ::=$ | constraint |
| $\varnothing$ | empty environment | $V : \mathsf{c}(\pi)$ | channel constraint |
| $C$ | constraint | $V\langle V\rangle$ | output link |
| $\Theta\,,\,\Xi$ | concatenation | $V(V)$ | input link |

As we have already seen, abstract values are simply the union of abstract names and abstract variables. They are used by constraint environments to refer to the bound names and bound variables of a system. For example, the system $(\nu n)(a[n\,\langle p\rangle])$ would be assigned the constraint environment $\mathsf{n}\langle p\rangle$ where $\mathsf{n}$ is an abstract name that refers to the bound name $n$ in the system and is used to distinguish it from any other bound or free name. We use $av(\Theta)$ to denote the set of abstract values present in the constraint environment $\Theta$ and give its formal definition below.

**Definition 6.4.** Given a constraint environment $\Theta$, the set of abstract values occurring in it is defined as follows:

$$av(\varnothing) \triangleq \emptyset \qquad av(\Theta\,,\Xi) \triangleq av(\Theta) \cup av(\Xi)$$
$$av(V:\pi) \triangleq av(V) \qquad av(V\langle V'\rangle) \triangleq av(V) \cup av(V') \qquad av(V(V')) \triangleq av(V) \cup av(V')$$
$$av(\mathsf{n}) \triangleq \{\mathsf{n}\} \qquad av(\mathsf{x}) \triangleq \{\mathsf{x}\} \qquad av(p) \triangleq \emptyset \qquad av(x) \triangleq \emptyset$$

Since abstract values are used to refer to bound names and bound variables, they induce a notion of $\alpha$-equivalence in constraint environments. We denote this by $\equiv_{c\alpha}$. For example, the environment $m\langle\mathsf{x}\rangle$ can be $\alpha$-converted to $m\langle\mathsf{y}\rangle$. We extend this to a structural congruence relation by admitting changes to the position of constraint mappings within a constraint environment. We use $\equiv_c$ to denote the structural congruence relation on constraint environments. Moreover, we also define the notion of *sub-environment*, written $\sqsubseteq_c$, to act as a partial order between constraint environments. This will be useful later in the proofs.

**Definition 6.5.** A constraint environment $\Theta$ is said to be a sub-environment of $\Xi$, written $\Theta \sqsubseteq_c \Xi$ if there exists an environment $\Xi'$ such $\Theta \subseteq \Xi'$ and $\Xi' \equiv_c \Xi$.

**Typing contexts.** The composition of a constraint environment and a type environment gives us a complete characterisation of the context in which a system is to be run and allows us to type check it separately from its context. We refer to this composition of a constraint environment and a type environment as a *typing context*. We denote by $\Theta \circ \Gamma$ the typing context composed of constraint environment $\Theta$ and type environment $\Gamma$. We also use $\varnothing$ as a shorthand for the empty typing context $\varnothing \circ \varnothing$ composed of the empty constraint environment and the empty type environment.

### 6.2.1.3 Aliases and constraints

**Aliases.** The previous example demonstrated the need for output and input links in constraint environments. In the example, the links $m\langle n\rangle$ and $m(x)$ were used to represent the fact that name $n$ is sent on channel $m$ by some principal, and then received by another one to be substituted for variable $x$. This meant that any outputs on channel $n$ needed to satisfy the constraints of variable $x$. Although not shown in the example, links work in the opposite direction as well. An output on some variable needs to satisfy the constraints of all channels that may be substituted for the variable. To make it easier to work with links in the type system, we define the notion of *aliases*. The aliases of a value are all the values that may end up being equivalent to it at run time, either a channel name and all the variables that it may be substituted for, a variable and all the values that may be substituted for it, or any values related by the transitive closure of the two. The aliases of a value are defined with respect to a particular constraint environment. We use $\mathit{aliases}_{\Theta}(V)$ to denote the set of aliases of $V$ with respect to constraint environment $\Theta$. The formal definition of the function $\mathit{aliases}_{\Theta}(-)$ is given in Figure 6.4.

Figure 6.4. *Definition of* $\mathit{aliases}_{\Theta}(V)$

TLink Forward

$$\frac{V''\langle V\rangle \in \Theta \quad V''(V') \in \Theta}{V' \in \mathit{aliases}_{\Theta}(V)}$$

TLink Backward

$$\frac{V''\langle V'\rangle \in \Theta \quad V''(V) \in \Theta}{V' \in \mathit{aliases}_{\Theta}(V)}$$

TLink Closure

$$\frac{V' \in \mathit{aliases}_{\Theta}(V'') \quad V'' \in \mathit{aliases}_{\Theta}(V)}{V' \in \mathit{aliases}_{\Theta}(V)}$$

**Constraints.** An output on a particular channel is required to satisfy the patterns associated with the channel itself as well as those associated with any of its aliases. Therefore, for the purposes of accounting for the constraints of a value, we can consider the set of aliases of a channel as forming an equivalence class. This is reflected in the definition of the function $\mathit{constraints}_{\Theta}(-)$ given below, which includes the patterns that are associated with the value explicitly as well as those associated with any of its aliases.

**Definition 6.6** (Constraints of a channel)**.** The set $\mathit{constraints}_{\Theta}(V)$ of constraints of a channel $V$ with respect to a constraint environment $\Theta$ is defined as follows:

$$\mathit{constraints}_{\Theta}(V) \triangleq \{\pi \mid V : \pi \in \Theta\} \cup \mathit{constraints}_{\Theta}(V') \text{ for all } V' \in \mathit{aliases}_{\Theta}(V)$$

### 6.2.1.4 Typing rules

The type and effect system has two kinds of type judgements, a primary one for systems and an auxiliary one for processes. These are given below:

- $\Theta \circ \Gamma \vdash S \triangleright \Xi$ which may be read as "system $S$ is well-typed with respect to typing context $\Theta \circ \Gamma$ and gives rise to its own constraint environment $\Xi$".

- $\Theta \circ \Gamma \vdash_a P \triangleright \Xi$ which may be read as "process $P$ is well-typed at principal $a$ with respect to typing context $\Theta \circ \Gamma$ and gives rise to its own constraint environment $\Xi$".

The typing context $\Theta \circ \Gamma$ provides an abstraction of the context for the purposes of type checking. The type judgement $\Theta \circ \Gamma \vdash S \triangleright \Xi$ tells us that system $S$ respects all the constraints imposed by the constraint environment $\Theta$. This means we can plug it into any context *consistent* with $\Theta$ (and typing environment $\Gamma$) and obtain a well-typed system. So what does it mean for a system to be consistent with a constraint environment? The simplest answer to this can be found in the typing judgement itself; a system is consistent with the constraint environment it gives rise to. To see this in action, consider the rule for type checking the parallel composition of two systems, which can be given as follows:

$$\frac{\Theta\,,\Xi_2 \circ \Gamma \vdash S_1 \triangleright \Xi_1 \quad \Theta\,,\Xi_1 \circ \Gamma \vdash S_2 \triangleright \Xi_2}{\Theta \circ \Gamma \vdash S_1 \mid S_2 \triangleright \Xi_1\,,\Xi_2} \quad av(\Xi_1) \cap av(\Xi_2) = \emptyset$$

The rule shows that in order to be able to type check the parallel composition of $S_1$ and $S_2$, we need to first type check $S_1$ and $S_2$ separately. We can do this because the typing context each of them is checked against accounts for the constraints raised by the other. System $S_1$ is type checked against the constraint environment $\Theta \circ \Xi_2$, representing the composition of the constraint environment $\Theta$ obtained from the external context of $S_1 \mid S_2$ and the constraint environment $\Xi_2$ obtained from the system $S_2$. In turn, system $S_2$ is type checked against the constraint environment $\Theta \circ \Xi_1$, representing the composition of the constraint environment $\Theta$ obtained from the external context of $S_1 \mid S_2$ and the constraint environment $\Xi_1$ obtained from the system $S_1$. The side condition of the rule just ensures that there is no conflict between the abstract values of the two constraint environments $\Xi_1$ and $\Xi_2$. We will see later how the constraint environment of the system itself is defined from its structure. It is also worth noting that consistency allows for systems that impose less constraints. That is, as expected, if we find that a system is well-typed with respect to a particular constraint environment $\Theta$, then we should be able to plug it into any context that imposes the same or less constraints than those of the constraint environment $\Theta$.

The typing judgements are defined by the rules of Figure 6.5. There are two sets of rules; one for systems which defines judgements of the form $\Theta \circ \Gamma \vdash S \triangleright \Xi$, and one for processes which defines judgements of the form $\Theta \circ \Gamma \vdash_a P \triangleright \Xi$. In the following we explain the rules in each set.

Figure 6.5. *Typing rules*

**Typing rules for systems.**

TSys Prin
$$\frac{\Theta \circ \Gamma \vdash_a P \triangleright \Xi}{\Theta \circ \Gamma \vdash a[P] \triangleright \Xi}$$

TSys Msg
$$\frac{\|w\|_\Gamma \models \mathit{constraints}_\Theta(m)}{\Theta \circ \Gamma \vdash m\langle\!\langle w \rangle\!\rangle \triangleright m\langle |w| \rangle}$$

TSys Res
$$\frac{\Theta \circ \Gamma \vdash S \triangleright \Xi\{^n/_{\mathsf{n}}\}}{\Theta \circ \Gamma \vdash (\nu n)S \triangleright \Xi}$$

TSys Par
$$\frac{\Theta\,,\Xi_2 \circ \Gamma \vdash S_1 \triangleright \Xi_1 \quad \Theta\,,\Xi_1 \circ \Gamma \vdash S_2 \triangleright \Xi_2}{\Theta \circ \Gamma \vdash S_1 \mid S_2 \triangleright \Xi_1\,,\Xi_2} \quad av(\Xi_1) \cap av(\Xi_2) = \emptyset$$

TSys Nil
$$\frac{}{\Theta \circ \Gamma \vdash 0 \triangleright \varnothing}$$

**Typing rules for processes.**

TProc Out
$$\frac{a!\|w\|_\Gamma\,;\,\|w'\|_\Gamma \models \mathit{constraints}_\Theta(|w|)}{\Theta \circ \Gamma \vdash_a w\,\langle w' \rangle \triangleright |w|\langle |w'| \rangle}$$

TProc In
$$\frac{\Theta\,,|w|(\mathsf{x}) \circ \Gamma\,,x : a?\|w\|_\Gamma\,;\pi \vdash_a P \triangleright \Xi\{^x/_{\mathsf{x}}\}}{\Theta \circ \Gamma \vdash_a w\,(\pi \text{ as } x).P \triangleright |w|(\mathsf{x})\,,|w| : \mathsf{c}(\pi)\,,\Xi}$$

TProc Par
$$\frac{\Theta\,,\Xi_2 \circ \Gamma \vdash_a P_1 \triangleright \Xi_1 \quad \Theta\,,\Xi_1 \circ \Gamma \vdash_a P_2 \triangleright \Xi_2}{\Theta \circ \Gamma \vdash_a P_1 \mid P_2 \triangleright \Xi_1\,,\Xi_2} \quad av(\Xi_1) \cap av(\Xi_2) = \emptyset$$

TProc Res
$$\frac{\Theta \circ \Gamma \vdash_a P \triangleright \Xi\{^n/_{\mathsf{n}}\}}{\Theta \circ \Gamma \vdash_a (\nu n)P \triangleright \Xi}$$

TProc Rep
$$\frac{\Theta \circ \Gamma \vdash_a P \triangleright \Xi}{\Theta \circ \Gamma \vdash_a *P \triangleright \Xi}$$

TProc Nil
$$\frac{}{\Theta \circ \Gamma \vdash_a 0 \triangleright \varnothing}$$

TProc Mat
$$\frac{\Theta \circ \Gamma \vdash_a P_1 \triangleright \Xi_1 \qquad \Theta \circ \Gamma \vdash_a P_2 \triangleright \Xi_2}{\Theta \circ \Gamma \vdash_a \text{if } w_1 = w_2 \text{ then } P_1 \text{ else } P_2 \ \triangleright \Xi_1\,,\Xi_2}$$

**Type checking systems.** The simplest rule in this set is that for the nil system given by TSys Nil. This rule simply says that the nil system 0, having no behaviour, is well-typed with respect to any typing context $\Theta \circ \Gamma$ and imposes no constraints of its own. The rule TSys Prin is also quite simple; it states that the single principal $a[P]$ is well typed under typing context $\Theta \circ \Gamma$ if the process $P$ is well-typed at principal $a$ with respect to the same typing context. The rules for type checking processes are explained in the next paragraph. The rule for type checking messages is given by TSys Msg. Intuitively, this says that in order for the message $m\langle\!\langle w \rangle\!\rangle$ to be well-typed, the provenance of value $w$ needs to satisfy any constraints imposed on channel $m$. This is what is expressed by the premise of the rule $\|w\|_\Gamma \models \mathit{constraints}_\Theta(m)$. The only constraint that the message $m\langle\!\langle w \rangle\!\rangle$ gives rise to is the output link $m\langle |w| \rangle$. Note that $|{-}|$ is the annotation erasure function defined by Definition 4.3 in Chapter 4. The rule for restriction, TSYS RES, states that the restricted system $(\nu n)S$ is well-typed with respect to typing context $\Theta \circ \Gamma$ if the system $S$ is well-typed with respect to the same context. What is interesting about this rule is the constraint environment that results from the system $(\nu n)S$. Let $\Xi$ stand for this constraint environment. We know that this constraint environment has to refer to any channels involved in inputs or outputs in $(\nu n)S$, including possibly the bound channel $n$ itself. To be able to refer to the channel name $n$, the constraint environment $\Xi$ has to use the abstract name $\mathsf{n}$. Therefore, the constraint environment of system $S$ where the name $n$ appears free has to be $\Xi\{^n/_\mathsf{n}\}$, the same as the constraint environment of $(\nu n)S$ but with the name $n$ substituted for all occurrences of the abstract name $\mathsf{n}$. We have already described the typing rule for parallel composition which is given by TSYS PAR.

**Type checking processes.** The type judgement for processes takes the form $\Theta \circ \Gamma \vdash_a P \triangleright \Xi$. The first thing to note about this judgement when compared to that of systems $\Theta \circ \Gamma \vdash S \triangleright \Xi$ is that processes are type checked with respect to a particular principal. This means that a process $P$ may be well-typed with respect to principal $a$ but not with respect to principal $b$. The reason for this should be self-evident, the type system is checking that the provenance of values would always satisfy the provenance policies of principals. Since the provenance of data is made up primarily of principal names, it is expected that changing these would affect the satisfiability of provenance policies. The two main rules for type checking processes are those for output TPROC OUT and input TPROC IN. The rule for output states that the process $w\,\langle w' \rangle$ located at principal $a$ would be well-typed with respect to typing context $\Theta \circ \Gamma$ if the provenance sequence $a!\|w\|_\Gamma\,;\|w'\|_\Gamma$ satisfies the constraint set given by $\mathit{constraints}_\Theta(|w|)$. The sequence $a!\|w\|_\Gamma\,;\|w'\|_\Gamma$ is what we would expect the provenance of value $w'$ to be after it is sent on channel $w$; the principal sending the value is $a$, the provenance of the channel is approximated by $\|w\|_\Gamma$, and the previous provenance of the value is approximated by $\|w'\|_\Gamma$. The channel that this value is being sent on is $w$ and therefore its constraints are given by $\mathit{constraints}_\Theta(|w|)$. The process $w\,\langle w' \rangle$ gives rise to the output link $|w|\langle |w'| \rangle$. The rule for type checking the input process $w\,(\pi\ \mathsf{as}\ x).P$ located at principal $a$ states that for this to be well-typed under the typing context $\Theta \circ \Gamma$, the process $P$ located at the same principal needs to be well-typed under the context $\Theta\,,|w|(\mathsf{x}) \circ \Gamma\,, x : a?\|w\|_\Gamma\,;\pi$. Note that both the constraint environment as well as the type environment have changed. The constraint environment has been extended with the input link $|w|(\mathsf{x})$ to account for the input that has just taken place, while the type environment has been extended with the type assignment $x{:}a?\|w\|_\Gamma;\pi$ to account for the type of variable $x$. The need to add the input link to the constraint environment is to detect any new aliases that might be created as a result of the input action. The variable $x$ which is bound in $w\,(\pi\ \mathsf{as}\ x).P$ occurs free in $P$ and so we need to account for its type in the type environment. We expect the value to be substituted for variable $x$ to have provenance matching the pattern $\pi$ and so after updating its provenance to reflect the input action, we would expect it to have provenance of the form $a?\|w\|_\Gamma\,;\pi$. This latter approximates the provenance of the value to be received and states that the value was received by $a$ on a channel with provenance $\|w\|_\Gamma$ and that before this, it had provenance matching $\pi$. The input process $w\,(\pi\ \mathsf{as}\ x).P$ gives rise to constraint environment $|w|(\mathsf{x})\,, |w| : \mathsf{c}(\pi)\,, \Xi$; the input link $|w|(\mathsf{x})$ as we saw lets us detect any aliases that may be created as a result of the input, the channel constraint $|w| : \mathsf{c}(\pi)$ ensures that only data with the correct provenance is consumed, while the constraint environment $\Xi$ represents those constraints imposed by the continuation process $P$ (where occurrences of the bound variable $x$ are referenced using the abstract variable $\mathsf{x}$). The rule for replication, given by TProc Rep, simply states that the replicated process $*\,P$ is well-typed under the same conditions as the process $P$. Rule TProc Mat is used to type check matching $\mathsf{if}\ w_1 = w_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ ; it states that both $P_1$ and $P_2$ need to satisfy all constraints imposed by their context, and the context needs to satisfy the constraints imposed by either of them. Finally, the rules for parallel composition, restriction and the nil process are similar to their counterparts for systems.

### 6.2.2 Type preservation

Type preservation (or subject reduction) states that well-typedness is invariant under the rules of reduction. This is important for every type system as it ensures that once a system is proved well-typed, it is guaranteed to remain so in every state it evolves to, and therefore any safety guarantees offered for the original system also apply to any of its derivatives. The definition of well-typedness is simple; a system $S$ is well-typed under typing context $\Theta \circ \Gamma$ and has constraints $\Xi$ if the judgement $\Theta \circ \Gamma \vdash S \triangleright \Xi$ is derivable using the typing rules of Figure 6.5. Often, we will be dealing with closed systems (i.e. systems with no free variables) and therefore we would expect the type environment to be empty. In fact, if we are simply interested in type checking the system itself in isolation with no requirement to plug it into a larger context, then the constraint environment of the context can be dropped too. Therefore, the definition of well-typedness can be simplified to the ability to derive the judgement $\Theta \circ \Gamma \vdash S \triangleright \Xi$ for any typing context $\Theta \circ \Gamma$, including possibly the empty context $\varnothing$. This can be simplified further by observing that the constraints of the system $S$ itself, given by the constraint environment $\Xi$, would also be irrelevant in such cases. We use $\vdash S \triangleright \diamond$, to be read simply as "S is well-typed", to mean that there exists a typing context $\Theta \circ \Gamma$ and constraint environment $\Xi$ such that $\Theta \circ \Gamma \vdash S \triangleright \Xi$. The definition of well-typedness is given in 6.7.

**Definition 6.7** (Well-typedness)**.** We say that a system $S$ is well-typed under typing context $\Theta \circ \Gamma$ and has constraints $\Xi$ if the judgement $\Theta \circ \Gamma \vdash S \triangleright \Xi$ can be derived using the rules in Figure 6.5. We say that a system $S$ is well-typed, written $\vdash S \triangleright \diamond$, if it is well-typed under some typing context $\Theta \circ \Gamma$ and has some constraint environment $\Xi$.

As we mentioned already, aliases behave like equivalence classes with respect to constraints. This fact is exhibited in the two properties of aliases given in lemmas 6.8 and 6.9. The first, Lemma 6.8, states that the set of constraints of a value $V$ are the same as those of any of its aliases. The second, Lemma 6.9 states that the set of constraints of a value with respect to a constraint environment remains the same if we substitute a value with any of its aliases in that constraint environment. These lemmas as well the subsequent ones are needed in the proof of type preservation given in Theorem 6.16.

**Lemma 6.8.** *if* $V' \in \mathit{aliases}_\Theta(V)$ *then* $\mathit{constraints}_\Theta(V) = \mathit{constraints}_\Theta(V')$

*Proof.* You will recall from the definition of $\mathit{constraints}_\Theta(-)$ that $\mathit{constraints}_\Theta(V) \triangleq \{\pi \mid V : \pi \in \Theta\} \cup \mathit{constraints}_\Theta(V')$ for all $V' \in \mathit{aliases}_\Theta(V)$. Let $\pi$ be a pattern such that $\pi \in \mathit{constraints}_\Theta(V')$. Then it follows from the definition that $\pi \in \mathit{constraints}_\Theta(V)$. Since by definition, $V' \in \mathit{aliases}_\Theta(V)$ implies $V \in \mathit{aliases}_\Theta(V')$, then it also follows that for all patterns $\pi$, if $\pi \in \mathit{constraints}_\Theta(V)$ then $\pi \in \mathit{constraints}_\Theta(V')$. Therefore, we conclude that $\mathit{constraints}_\Theta(V) = \mathit{constraints}_\Theta(V')$ □

**Lemma 6.9.** *if* $V' \in \mathit{aliases}_\Theta(V)$ *then* $\mathit{constraints}_{\Theta\{V'/V\}}(V'') = \mathit{constraints}_\Theta(V'')$

*Proof.* Assume that $V' \in \mathit{aliases}_\Theta(V)$. Lemma 6.8 implies that $\mathit{constraints}_\Theta(V) = \mathit{constraints}_\Theta(V')$. If we consider the constraints of $V''$ with respect to $\Theta\{^{V'}/_V\}$, we find there are two cases, either $V'$ and $V$ are aliases of $V''$ or not. If $V'$ and $V$ are aliases of $V''$, then substituting $V'$ for $V$ should yield $\mathit{constraints}_{\Theta\{V'/V\}}(V'') = C \cup \mathit{constraints}_{\Theta\{V'/V\}}(V')$ for some set of patterns $C$. We can show that $\mathit{constraints}_\Theta(V'') = C \cup \mathit{constraints}_\Theta(V')$ and since we know that $\mathit{constraints}_\Theta(V) = \mathit{constraints}_\Theta(V')$, then we can conclude that $\mathit{constraints}_{\Theta\{V'/V\}}(V'') = \mathit{constraints}_\Theta(V'')$. If $V$ and $V'$ are not aliases of $V''$ then they should not have any impact on the set of aliases of $V''$ and therefore its set of constraints will also remain the same. □

Lemma 6.10, commonly referred to as the Substitution Lemma in the type systems literature, states that we may replace a variable in a system with any of its aliases, as long as the alias has the same type or a stronger one. The two conditions are important to ensure this holds. The first condition, the value being substituted for the variable must be in the variable's set of aliases, ensures that the substitution has no impact on the sets of constraints entailed by the constraint environment. The second condition, requiring the type of the value to be at least as strong as that assigned to the variable, ensures that it will still satisfy any constraints that were previously satisfied by the type of the variable.

**Lemma 6.10** (Substitution Lemma)**.** *if* $\Theta \circ \Gamma, x : t \vdash_a P \triangleright \Xi$*,* $V \in \mathit{aliases}_\Theta(x)$ *and* $t' \preceq t$ *then* $\Theta \circ \Gamma \vdash_a P\{^{V:t'}/_x\} \triangleright \Xi\{^V/_x\}$*.*

*Proof.* Let us assume that $\Theta \circ \Gamma, x{:}t \vdash_a P \triangleright \Xi$, $V \in \mathit{aliases}_\Theta(x)$ and $t' \preceq t$ all hold. We need to show that $\Theta \circ \Gamma \vdash_a P\{^{V:t'}/_x\} \triangleright \Xi\{^V/_x\}$ follows from these assumptions. We do this by induction on the derivation of $\Theta \circ \Gamma, x{:}t \vdash_a P \triangleright \Xi$. There are 7 cases in total corresponding to the 7 rules for type checking processes. The details are tedious but straightforward. The only point worth noting is the importance of the two conditions $V \in \mathit{aliases}_\Theta(x)$ and $t' \preceq t$. The condition $V \in \mathit{aliases}_\Theta(x)$ ensures that the substitution does not change the set of constraints of any channel name, while the condition $t' \preceq t$ guarantees that any satisfaction involving the type $t$ will still hold after the substitution. □

When a system is declared well-typed with respect to a particular constraint environment, that guarantees that it satisfies all the constraints raised by that constraint environment. It should be expected then that this same system would be well-typed with respect to any subset of those constraints since any such subset would have at most the same constraints. This is proved in Lemma 6.12. Along the same lines, Lemma 6.13 states that any subsystem of a given system would be guaranteed to be well-typed with respect to any constraint environment that the system itself is well-typed with respect to. The reason is that given a system, any of its subsystems would have the same outputs as the system itself or even less. Therefore, if the system is well-typed with respect to a particular constraint environment, that means all its outputs must satisfy the constraints implied by the constraint environment. Hence, any subset of these outputs must also satisfy those same constraints. Lemma 6.11 is useful for the proof of Lemma 6.12. It simply states that larger constraint environments would give rise to more constraints than smaller ones.

**Lemma 6.11.** *if* $\Theta \sqsubseteq_c \Xi$ *then* $\mathit{constraints}_\Theta(m) \subseteq \mathit{constraints}_\Xi(m)$

*Proof.* It's sufficient to observe that, by definition, $\mathit{constraints}_\Xi(m)$ is simply the set of constraints imposed on channel $m$ or on one of its aliases by environment $\Xi$. Environment

$\Theta$, being a sub-environment of $\Xi$, would impose less constraints on channel $m$ and its aliases, and may even include less aliases for $m$. Therefore, the set inclusion clearly holds. □

**Lemma 6.12** (Context Weakening). *if* $\Theta\, ,\, \Theta' \circ \Gamma \vdash S \triangleright \Xi$ *then* $\Theta \circ \Gamma \vdash S \triangleright \Xi$

*Proof.* The idea behind the proof is to observe that ultimately, the well-typedness of a system boils down to showing that its outputs respect all the constraints of their channels. In the case of the first type judgement $\Theta\, ,\, \Theta' \circ \Gamma \vdash S \triangleright \Xi$, this means proving satisfactions of the form $t \models \mathit{constraints}_{\Theta,\Theta'}(m)$. By Lemma 6.11, any such satisfaction implies $t \models \mathit{constraints}_{\Theta}(m)$. This in turn implies that the type judgement $\Theta \circ \Gamma \vdash S \triangleright \Xi$ must hold. □

**Lemma 6.13** (System Weakening). *if* $\Theta \circ \Gamma \vdash S \mid S' \triangleright \Xi\, ,\, \Xi'$ *then* $\Theta \circ \Gamma \vdash S \triangleright \Xi$

*Proof.* The proof for this one is similar to the proof of Context Weakening. Since $\Theta \circ \Gamma \vdash S \mid S' \triangleright \Xi\, ,\, \Xi'$ holds, then every output in $S \mid S'$ must satisfy all the constraints raised by the constraint environment $\Theta$. The outputs of $S$ are a subset of those of $S \mid S'$ and hence it follows that they satisfy the constraints raised by $\Theta$. Hence, it follows that $\Theta \circ \Gamma \vdash S \triangleright \Xi$ holds. □

The Subject Congruence lemma, given as Lemma 6.14, states that structurally congruent systems are all well-typed under the same conditions. Again, this is to be expected as the type system aims to ensure that all provenance policies raised by principals of a system are guaranteed to be satisfied by all outputs of the system. Structurally congruent systems are essentially the same and only differ in syntax. Hence, they have the same behaviour; their principals impose the same provenance policies and make the same outputs.

**Lemma 6.14** (Subject Congruence). *if* $\Theta \circ \Gamma \vdash S \triangleright \Xi$ *and* $S \equiv_a S'$ *then* $\Theta \circ \Gamma \vdash S' \triangleright \Xi'$ *for some* $\Xi'$ *such that* $\Xi \equiv_c \Xi'$.

*Proof.* Let us assume that $\Theta \circ \Gamma \vdash S \triangleright \Xi$ and $S \equiv_a S'$ hold. The proof of $\Theta \circ \Gamma \vdash S' \triangleright \Xi'$ where $\Xi'$ is a constraint environment such that $\Xi \equiv_c \Xi'$ proceeds by induction on the last rule used in the derivation of $S \equiv_a S'$. The details are tedious but straightforward and are therefore omitted. □

Lemma 6.15 is a stronger version of Type Preservation given in Theorem 6.16. It states that if a system is well-typed with respect to a typing context, then any of its derivatives must be well-typed with respect to the same typing context. Two points should be noted here. The first is that the typing context stays the same after the transition ensuring that the system will continue to comply with any constraints imposed by the external environment. The second point to note is that the internal constraint environment of the system itself stays the same or weakens. This guarantees that it will at most retain the same constraints which ensures that it may still be plugged into any external environment that it could be plugged into before the transition.

**Lemma 6.15.** *if* $\Theta \circ \Gamma \vdash S \triangleright \Xi$ *and* $S \rightarrow_s S'$ *then* $\Theta \circ \Gamma \vdash S' \triangleright \Xi'$ *for some* $\Xi'$ *such that* $\Xi' \sqsubseteq_c \Xi$

*Proof.* Let us assume that $\Theta \circ \Gamma \vdash S \triangleright \Xi$ and $S \rightarrow_s S'$ hold. Now we need to prove that $\Theta \circ \Gamma \vdash S' \triangleright \Xi'$ also holds for some $\Xi'$ such that $\Xi' \sqsubseteq_c \Xi$. We do this by induction on the last rule used in the derivation of $S \rightarrow_s S'$. Let us illustrate some of the cases below.

Case SRed Snd. This means that $S \rightarrow_s S'$ is of the form $a[m{:}\kappa_m \langle p{:}\kappa_p\rangle] \rightarrow_s m\langle\!\langle p : a!\kappa_m\, ; \kappa_p\rangle\!\rangle$. From the assumption, we know that $\Theta \circ \Gamma \vdash a[m{:}\kappa_m \langle p{:}\kappa_p\rangle] \triangleright m\langle p\rangle$ . This latter must have been derived using the rules of Figure 6.5. There is only one possible derivation tree for this and it is as follows:

$$\cfrac{\cfrac{a!\kappa_m\, ; \kappa_p \models \mathit{constraints}_\Theta(m)}{\Theta \circ \Gamma \vdash_a m{:}\kappa_m \langle p{:}\kappa_p\rangle \triangleright m\langle p\rangle}}{\Theta \circ \Gamma \vdash a[m{:}\kappa_m \langle p{:}\kappa_p\rangle] \triangleright m\langle p\rangle}$$

The above derivation tree implies that $a!\kappa_m\, ; \kappa_p \models \mathit{constraints}_\Theta(m)$. Now, if we look at $S'$, that is $m\langle\!\langle p : a!\kappa_m\, ; \kappa_p\rangle\!\rangle$, and consider what we need to do to prove that $\Theta \circ \Gamma \vdash S' \triangleright \Xi'$ holds, we get the following:

$$\frac{a!\kappa_m\, ; \kappa_p \models \mathit{constraints}_\Theta(m)}{\Theta \circ \Gamma \vdash m\langle\!\langle m{:}a!\kappa_m\, ; \kappa_p\rangle\!\rangle \triangleright m\langle p\rangle}$$

Since we already know that $a!\kappa_m\, ; \kappa_v \models \mathit{constraints}_\Theta(m)$ holds then we can conclude that $\Theta \circ \Gamma \vdash m\langle\!\langle m{:}a!\kappa_m\, ; \kappa_p\rangle\!\rangle \triangleright m\langle p\rangle$ holds too. Note that the constraint environments of $S$ and $S'$ in this case are both $m\langle p\rangle$, and hence the sub-environment condition holds too.

Case SRed Rcv. This means that $S \rightarrow_s S'$ is of the form $a[m{:}\kappa_m (\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_s a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}]$. By assumption, we know that $\Theta \circ \Gamma \vdash a[m{:}\kappa_m (\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \triangleright m(x)\, , m : \mathsf{c}(\pi)\, , \Xi_P\, , m\langle p\rangle$ holds. The derivation for this must be of the form:

$$\frac{\Theta\, , \Xi_2 \circ \Gamma \vdash a[m{:}\kappa_m (\pi \text{ as } x).P] \triangleright \Xi_1 \qquad \Theta\, , \Xi_1 \circ \Gamma \vdash m\langle\!\langle v : \kappa_v\rangle\!\rangle \triangleright \Xi_2}{\Theta \circ \Gamma \vdash a[m{:}\kappa_m (\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \triangleright \Xi_1\, , \Xi_2}$$

where $\Xi_1$ and $\Xi_2$ are defined as follows:

$$\Xi_1 = m(x)\,,\, m : \mathsf{c}(\pi)\,,\, \Xi_P \qquad \Xi_2 = m\langle p\rangle$$

In the above, $\Xi_P$ is the constraint environment of process $P$. The derivation of the system $a[m{:}\kappa_m\,(\pi \text{ as } x).P]$ is as follows:

$$\cfrac{\cfrac{\Theta\,,\, m\langle p\rangle\,,\, m(x) \circ \Gamma\,,\, x : a?\kappa_m\,;\pi \vdash_a P \rhd \Xi_P}{\Theta\,,\, m\langle p\rangle \circ \Gamma \vdash_a m{:}\kappa_m\,(\pi \text{ as } x).P \rhd m(x)\,,\, m : \mathsf{c}(\pi)\,,\, \Xi_P}}{\Theta\,,\, m\langle p\rangle \circ \Gamma \vdash a[m{:}\kappa_m\,(\pi \text{ as } x).P] \rhd m(x)\,,\, m : \mathsf{c}(\pi)\,,\, \Xi_P}$$

while that of the system $m\langle\!\langle p : \kappa_p\rangle\!\rangle$ is as follows:

$$\cfrac{\kappa_p \models \mathit{constraints}_{\Theta,m(x),m:\mathsf{c}(\pi),\Xi_P}(m)}{\Theta\,,\, m(x)\,,\, m : \mathsf{c}(\pi)\,,\, \Xi_P \circ \Gamma \vdash m\langle\!\langle p : \kappa_p\rangle\!\rangle \rhd m\langle p\rangle}$$

To prove that $\Theta \circ \Gamma \vdash a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}] \rhd \Xi'$ holds for some $\Xi'$ such that $\Xi' \sqsubseteq_c \Xi$, we consider its derivation tree:

$$\cfrac{\Theta \circ \Gamma \vdash_a P\{^{p:a?\kappa_m;\kappa_p}/_x\} \rhd \Xi'}{\Theta \circ \Gamma \vdash a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}] \rhd \Xi'}$$

which means that we need to show that $\Theta \circ \Gamma \vdash_a P\{^{p:a?\kappa_m;\kappa_p}/_x\} \rhd \Xi'$ holds. From $\Theta\,, m\langle p\rangle\,,$ $m(x) \circ \Gamma, x : a?\kappa_m\,;\pi \vdash_a P \rhd \Xi_P$ (which holds by assumption as we saw), it follows $p$ is an alias of $x$. Since $\kappa_p \models \pi$, it follows that $a?\kappa_m\,;\kappa_p$ is a subtype of $a?\kappa_m\,;\pi$. Therefore, by the Substitution Lemma, we get $\Theta\,,\, m\langle p\rangle\,,\, m(x) \circ \Gamma \vdash_a P\{^{p:a?\kappa_m;\kappa_p}/_x\} \rhd \Xi_P\{^p/_x\}$. From this latter, using Lemma 6.12, we deduce that $\Theta \circ \Gamma \vdash_a P\{^{p:a?\kappa_m;\kappa_p}/_x\} \rhd \Xi_P\{^p/_x\}$. Therefore, we may conclude that $\Theta \circ \Gamma \vdash a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}] \rhd \Xi_P\{^p/_x\}$ holds. Note that indeed, $\Xi_P\{^p/_x\}$ is a sub-environment of $m(x)\,,\, m : \mathsf{c}(\pi)\,,\, \Xi_P$.

Case SRed Res. This means that the last two lines of the derivation were as follows:

$$\cfrac{T \longrightarrow_s T'}{(\nu n)T \longrightarrow_s (\nu n)T'}$$

where $S = (\nu n)T$ and $S' = (\nu n)T'$ for some $T$ and $T'$. We know that, by assumption, $\Theta \circ \Gamma \vdash (\nu n)T \rhd \Xi$ holds. This judgement must have been derived using the TSys Res rule as follows:

$$\cfrac{\Theta \circ \Gamma \vdash T \rhd \Xi\{^n/_{\mathsf{n}}\}}{\Theta \circ \Gamma \vdash (\nu n)T \rhd \Xi}$$

From $\Theta \circ \Gamma \vdash T \rhd \Xi\{^n/_{\mathsf{n}}\}$ and $T \longrightarrow_s T'$, we deduce by the induction hypothesis that $\Theta \circ \Gamma \vdash T' \rhd \Xi'\{^n/_{\mathsf{n}}\}$ for some $\Xi'$ such that $\Xi' \sqsubseteq_c \Xi$. From $\Theta \circ \Gamma \vdash T' \rhd \Xi'\{^n/_{\mathsf{n}}\}$, and by applying the rule TSys Res, we may conclude that $\Theta \circ \Gamma \vdash (\nu n)T' \triangleright \Xi'$. Again, note that $\Xi' \sqsubseteq_c \Xi$. □

**Theorem 6.16** (Subject Reduction)**.** *if* $\vdash S \triangleright \diamond$ *and* $S \rightarrow_s S'$ *then* $\vdash S' \triangleright \diamond$.

*Proof.* Follows directly from the definition of $\vdash S \triangleright \diamond$ and Lemma 6.15. □

### 6.2.3 Type safety

Type safety, formulated below in Theorem 6.17, states that all reductions of a well-typed system $S$ which are derivable using the statically typed reduction relation $\rightarrow_s$ are also derivable using the dynamically typed one $\rightarrow_a$. What this means is that despite the absence of dynamic provenance checks, the relation $\rightarrow_s$, when restricted to well-typed systems, would not admit any new system transitions compared to $\rightarrow_a$. In particular, it would never admit any *unsafe* transitions that might otherwise lead to principals consuming data with provenance that violates their policies. The implication of this of course is that dynamic provenance checking is redundant for well-typed systems.

**Theorem 6.17** (Type Safety)**.** $\vdash S \triangleright \diamond$ *and* $S \rightarrow_s S'$ *implies* $S \rightarrow_a S'$.

*Proof.* Let us assume that $\vdash S \triangleright \diamond$ and $S \rightarrow_s S'$ hold. We need to show that $S \rightarrow_a S'$ follows from these two assumptions. We do this by induction on the last rule used in the derivation of $S \rightarrow_s S'$.

Case SRed Snd. This means that $S \rightarrow_s S'$ was of the form $a[m{:}\kappa_m\langle p{:}\kappa_p\rangle] \rightarrow_s m\langle\!\langle p : a!\kappa_m ; \kappa_p\rangle\!\rangle$. Now, by applying the rule ARed Snd, we get $a[m{:}\kappa_m\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p : a!\kappa_m ; \kappa_p\rangle\!\rangle$, which means that $S \rightarrow_a S'$.

Case SRed Rcv. This means that $S \rightarrow_s S'$ was of the form $a[m{:}\kappa_m(\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_s a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}]$. Since $S$ is well-typed, then we know that there exists some type context $\Theta \circ \Gamma$ and constraint environment $\Xi$ such that $\Theta \circ \Gamma \vdash a[m{:}\kappa_m(\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \triangleright \Xi$ holds. The derivation for this must have been as follows:

$$\dfrac{\dfrac{\Xi_2 \circ \emptyset \vdash_a m{:}\kappa_m(\pi \text{ as } x).P \triangleright \Xi_1}{\Xi_2 \circ \emptyset \vdash a[m{:}\kappa_m(\pi \text{ as } x).P] \triangleright \Xi_1} \qquad \dfrac{\kappa_p \models \mathit{constraints}_{\Xi_1}(m)}{\Xi_1 \circ \emptyset \vdash m\langle\!\langle p : \kappa_p\rangle\!\rangle \triangleright \Xi_2}}{\emptyset \circ \emptyset \vdash a[m{:}\kappa_m(\pi \text{ as } x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \triangleright \Xi_1 \circ \Xi_2}$$

where $\Xi_1$ and $\Xi_2$ are the constraint environments of $a[m{:}\kappa_m\,(\pi \;\mathsf{as}\; x).P]$ and $m\langle\!\langle p : \kappa_p\rangle\!\rangle$ respectively and are defined as:

$$\Xi_1 = m(x)\,,m : \mathsf{c}(\pi)\,,\Xi_1'$$
$$\Xi_2 = m\langle p\rangle$$

In the above, $\Xi_1'$ refers the constraint environment of process $P$. The above derivation implies that $\kappa_p \models \mathit{constraints}_{m(x),m:\mathsf{c}(\pi),\Xi_1}(m)$. We know that the constraints set of $m$ is given as $\mathit{constraints}_{m(x),m:\mathsf{c}(\pi),\Xi_1'}(m) = \pi \cup \mathit{constraints}_{\Xi_1'}(m)$. This implies that $\kappa_p \models \pi$. The transition $a[m{:}\kappa_m\,(\pi \;\mathsf{as}\; x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_a a[P\{^{p:a?\kappa_m;\kappa_p}/_x\}]$ could only be derived using the rule ARed Rcv if the premise $\kappa_p \models \pi$ holds, which we have already shown to be true. Hence, we conclude that $S \rightarrow_a S'$.

Case SRed Res. This means that the last two lines of the derivation were as follows:

$$\frac{T \rightarrow_s T'}{(\nu n)T \rightarrow_s (\nu n)T'}$$

where $S = (\nu n)T$ and $S' = (\nu n)T'$ for some $T$ and $T'$ respectively. Since $\vdash (\nu n)T \triangleright \diamond$ holds (by assumption), then using rule TSys Res, it is possible to infer that $\vdash T \triangleright \diamond$ holds too. Now, since we have that $\vdash T \triangleright \diamond$ and $T \rightarrow_s T'$, then we can use the induction hypothesis to get $T \rightarrow_a T'$. Rule ARed Res gives us $(\nu n)T \rightarrow_a (\nu n)T'$, and hence we may conclude that $S \rightarrow_a S'$.

Case SRed Par. This means that the last two lines of the derivation were as follows:

$$\frac{R \rightarrow_s R'}{R \mid T \rightarrow_s R' \mid T}$$

where $S = R \mid T$ and $S' = R' \mid T$ for some $R$, $T$ and $R'$. Since $S$ is well-typed, then $\vdash R \mid T \triangleright \diamond$. This latter judgement must have been derived using the rule TSys Par, which means that $\vdash R \triangleright \diamond$ and $\vdash T \triangleright \diamond$ are also true. Now, the fact that $\vdash R \triangleright \diamond$ and $R \rightarrow_s R'$, implies by the induction hypothesis that $R \rightarrow_a R'$. From this latter, we can derive $R \mid T \rightarrow_a R' \mid T$ using the rule ARed Par.

Case SRed Str. This means that the last two lines of the derivation were as follows:

$$\frac{S \equiv T \qquad T \rightarrow_s T' \qquad T' \equiv S'}{S \rightarrow_s S'}$$

We know by assumption that $\vdash S \triangleright \diamond$ holds. Since $S \equiv T$, by Lemma 6.14, this implies that $\vdash T \triangleright \diamond$. From this latter, and by the induction hypothesis, we can show that $T \rightarrow_a T'$.

Now since $S \equiv T$, $T \to_a T'$ and $T' \equiv S'$ all hold, using the rule ARED STR, we can show that $S \to_a S'$. □

Dropping the antecedent $\vdash S \triangleright \diamond$ would falsify the implication in the Type Safety theorem. The reason for this is that the statically typed reduction relation $\to_s$ lacks dynamic provenance checks and therefore, without type checking, it would admit unsafe transitions. A simple example of this would be the following system:

$$a[m\,\langle p\rangle] \mid b[m\,(c!\mathsf{Any}\ \mathsf{as}\ x).P]$$

This system would have the following transitions according to the relation $\to_s$:

$$\begin{aligned} a[m\,\langle p\rangle] \mid b[m\,(c!\mathsf{Any}\ \mathsf{as}\ x).P] \ &\to_s m\langle\!\langle p : a!\epsilon\rangle\!\rangle \mid b[m\,(c!\mathsf{Any}\ \mathsf{as}\ x).P] \\ &\to_s b[P\{^{p:b?\epsilon;a!\epsilon}/_x\}] \end{aligned}$$

where principal $b$ receives the value $p$ although it has provenance $a!\epsilon$ that violates its policy $c!\mathsf{Any}$. Type checking would detect this and declare this system as untypable.

The opposite implication, $S \to_a S' \implies (S \to_s S' \wedge \vdash S \triangleright \diamond)$, does not hold either. The reason is that with dynamic provenance checking in $\to_a$, principals are allowed to output data with provenance that violates some or even all provenance policies. These result in messages that "float around" in the system until they find a principal whose policy permits data with that provenance. Even if no such principal is found, the presence of these messages would not cause any safety issues. To illustrate this, consider the following system:

$$S \triangleq a[m\,\langle p\rangle] \mid b[m\,(a!\mathsf{Any}\ \mathsf{as}\ x)] \mid c[m\,(b!\mathsf{Any}\ \mathsf{as}\ y)]$$

This system is composed of three principals; principal $a$ produces value $p$ on channel $m$, while principals $b$ and $c$ are both ready to consume data on channel $m$. The provenance policies of $b$ and $c$ are different however. Principal $b$ only accepts data produced by $a$ whereas principal $c$ only accepts data produced by $b$. This means that $a$ is violating the provenance policy of $c$ by outputting data on $m$. Therefore, this system is not well-typed. Dynamic provenance checking would allow this system to make transitions, leading to principal $b$ receiving the value sent by $a$ as it has provenance that agrees with its policies. It would also guarantee that $c$ would never receive this value as it violates its provenance policy. This is illustrated by the following transitions of system $S$:

$$\begin{aligned} S \ &\to_a m\langle\!\langle p : a!\mathsf{Any}\rangle\!\rangle \mid b[m\,(a!\mathsf{Any}\ \mathsf{as}\ x).P] \mid c[m\,(b!\mathsf{Any}\ \mathsf{as}\ y).Q] \\ &\to_a b[P\{^{p:b?\mathsf{Any};a!\mathsf{Any}}/_x\}] \mid c[m\,(b!\mathsf{Any}\ \mathsf{as}\ y)] \end{aligned}$$

These examples demonstrate the difference between dynamic provenance checking and static provenance checking. In fact, these differences are inherent to dynamic and static type checking; dynamic type checking allows safe transitions to proceed while blocking unsafe ones, whereas static type checking takes an *all-or-nothing* approach where the system is only declared well-typed if *all* of its transitions are deemed safe. The following section looks at this more closely.

### 6.2.4 Discussion

Together, type preservation and type safety ensure that well-typed systems would never lead to principals consuming data with provenance that violates their trust policies. This means that dynamic provenance checks, as well as dynamic provenance tracking as we will see in the next section, may be dropped from well-typed systems without impacting the policies of principals. However, systems that fail type-checking will have to rely on dynamic provenance checking and tracking. The aim of this section is to look at the expressive power of the type system and analyse under what conditions would a system be well-typed or fail type checking.

Let us start with an example of a simple system $S$ defined as follows:

$$S \triangleq a[m{:}\kappa_{am}\,\langle p_a{:}\kappa_{ap}\rangle] \mid b[m{:}\kappa_{bm}\,\langle p_b{:}\kappa_{bp}\rangle] \mid c[m{:}\kappa_{cm}\,(\pi_{cm}\ \mathsf{as}\ x)] \mid d[m{:}\kappa_{dm}\,(\pi_{dm}\ \mathsf{as}\ y)]$$

System $S$ is composed of four principals; two producers $a$ and $b$, and two consumers $c$ and $d$. All four principals communicate on channel $m$. We leave the provenance of values and the policies of principals unspecified in this initial definition so that we can analyse how varying these would affect the well-typedness of the system. We use $A$, $B$, $C$ and $D$ to refer to the single principal systems composed of principals $a$, $b$, $c$ and $d$ respectively. So for example, $A$ refers to the single principal system $a[m{:}\kappa_{am}\,\langle p_a{:}\kappa_{ap}\rangle]$. Therefore, $S$ could be written simply as $A \mid B \mid C \mid D$. Starting with an empty typing context, the typing derivation of system $S$ would be as follows:

$$\dfrac{\dfrac{\dfrac{a!\kappa_{am}\,;\,\kappa_{ap} \models \mathit{constraints}_{\Xi_B,\Xi_C,\Xi_D}(m)}{\cdots}}{\Xi_B\,,\,\Xi_C\,,\,\Xi_D \circ \emptyset \vdash A \triangleright \Xi_A} \qquad \dfrac{\cdots}{\Xi_A\,,\,\Xi_C\,,\,\Xi_D \circ \emptyset \vdash B \triangleright \Xi_B} \qquad \cdots}{\emptyset \circ \emptyset \vdash A \mid B \mid C \mid D \triangleright \Xi_A\,,\,\Xi_B\,,\,\Xi_C\,,\,\Xi_D}$$

In the interest of space, we omit the type derivations for $C$ and $D$ and only show parts of the derivations of $A$ and $B$. The constraint environments $\Xi_A$, $\Xi_B$, $\Xi_C$ and $\Xi_D$ are those of systems $A$, $B$, $C$ and $D$ respectively and can be defined as follows:

$$\Xi_A = m\langle p_a\rangle \qquad \Xi_B = m\langle p_b\rangle \qquad \Xi_C = m(x)\,, m : \mathsf{c}(\pi_{cm}) \qquad \Xi_D = m(y)\,, m : \mathsf{c}(\pi_{dm})$$

Therefore, the premise $a!\kappa_{am}\ ;\ \kappa_{ap} \models \mathit{constraints}_{\Xi_B,\Xi_C,\Xi_D}(m)$ simplifies to $a!\kappa_{am}\ ;\ \kappa_{ap} \models \{\pi_{mc}, \pi_{md}\}$. Type checking $B$ also gives rise to a similar premise, and that in turn simplifies to $b!\kappa_{bm}\ ;\ \kappa_{bp} \models \{\pi_{mc}, \pi_{md}\}$. Systems $C$ and $D$, having no outputs, do not generate any such premises. Hence, for system $S$ to be well-typed, the following four conditions need to be satisfied:

$$a!\kappa_{am}\ ;\ \kappa_{ap} \models \pi_{mc} \qquad a!\kappa_{am}\ ;\ \kappa_{ap} \models \pi_{md} \qquad b!\kappa_{bm}\ ;\ \kappa_{bp} \models \pi_{mc} \qquad b!\kappa_{bm}\ ;\ \kappa_{bp} \models \pi_{md}$$

This basically says that the provenance of each of the two values sent by $a$ and $b$ needs to satisfy both of the policies imposed by $c$ and $d$. What this means is that all four principals have to agree on what the provenance of data communicated on channel $m$ should be, and to only use the channel in accordance with this. The calculus itself does not offer a mechanism to enforce such a global agreement on the use of channels. It does not need to though as with dynamic provenance checks, each principal can use the channel as they see fit without violating the provenance policies of other principals. The role of the static type system can be seen as analysing a system to infer if a global agreement on the use of channels has been reached, and if has, then the dynamic checks are declared redundant and may be dropped.

#### 6.2.4.1 Reconcilable provenance policies

For the system $S$ defined above to be well-typed, the policies $\pi_{cm}$ and $\pi_{dm}$ imposed on channel $m$ need to be *reconcilable*, that is, they need to impose similar conditions on the provenance of the data they are willing to consume on channel $m$. Moreover, the conditions they impose on channel $m$ need to be satisfied by the provenance of data that may be sent on channel $m$, namely $a!\kappa_{am}\ ;\ \kappa_{ap}$ and $b!\kappa_{bm}\ ;\ \kappa_{bp}$. Probably the simplest way to satisfy this is if the policies $\pi_{cm}$ and $\pi_{dm}$ are the same and are both equivalent to the pattern $\mathsf{Any}$. In fact, only one of the two patterns needs to be $\mathsf{Any}$ if the other one is satisfied by both provenance sequences. A simple pattern that satisfies both of the provenance sequences in this example is $a!\kappa_{am}\ ;\ \kappa_{ap} \vee b!\kappa_{bm}\ ;\ \kappa_{bp}$. It is easy to specify formally what it means for two or more patterns to be reconcilable as given in Definition 6.18.

**Definition 6.18** (Reconcilable patterns). Patterns $\pi_1, \ldots, \pi_n$ are said to be reconcilable if there exists a provenance type $t$ such that $t \models \{\pi_1, \ldots, \pi_n\}$. Patterns $\pi_1, \ldots, \pi_n$ are said to be reconcilable *with respect to* types $t_1, \ldots, t_m$ if for all $i \in 1..m$, it is the case that $t_i \models \{\pi_1, \ldots, \pi_n\}$.

Applying this definition to our example, for system $S$ to be well-typed, the patterns $\pi_{cm}$ and $\pi_{dm}$ need to be reconcilable with respect to the types $a!\kappa_{am} ; \kappa_{ap}$ and $b!\kappa_{bm} ; \kappa_{bp}$. If both $\pi_{cm}$ and $\pi_{dm}$ are equivalent to the pattern $\mathsf{Any}$, or if one of them is $\mathsf{Any}$ while the other is $a!\kappa_{am} ; \kappa_{ap} \vee b!\kappa_{bm} ; \kappa_{bp}$, then indeed the two patterns $\pi_{cm}$ and $\pi_{dm}$ are reconcilable with respect to the provenance types $a!\kappa_{am} ; \kappa_{ap}$ and $b!\kappa_{bm} ; \kappa_{bp}$.

#### 6.2.4.2 Typable and untypable systems

In the example of system $S$ we defined earlier, if the policies imposed by principals $c$ and $d$ on channel $m$ cannot be reconciled, then system $S$ would not be well-typed. For instance, the policies $\pi_{cm} \triangleq a!\mathsf{Any}$ and $\pi_{dm} \triangleq b!\mathsf{Any}$ cannot be reconciled since $\pi_{cm}$ states that only data originating from $a$ is acceptable, whereas $\pi_{dm}$ states that only data from $b$ is acceptable. This means that no provenance sequence can satisfy both $a!\mathsf{Any}$ and $b!\mathsf{Any}$ at the same time, and therefore, the only case under which any system with those two policies would be well-typed is if it had no outputs at all on channel $m$. However, in the case of system $S$, both $a$ and $b$ output on channel $m$ and therefore system $S$ is not well-typed when instantiated with those two policies. Let us assume that $\pi_{cm}$ and $\pi_{dm}$ are reconcilable. Note that all this implies is that there exists some provenance type that satisfies both of these policies, and so it is not enough on its own to guarantee system $S$ to be well-typed. For instance, the policies $\pi_{cm} \triangleq a!\mathsf{Any}$ and $\pi_{dm} \triangleq a!\mathsf{Any}$ are the same and therefore clearly reconcilable. Any system where only principal $a$ is allowed to output on channel $m$ would be well-typed in that case. However, system $S$ is not such a system as principal $b$ too produces data on channel $m$ and hence any data it outputs would not satisfy the pattern $a!\mathsf{Any}$. For system $S$ to be well-typed, the two patterns $\pi_{cm}$ and $\pi_{dm}$ need to be reconcilable with respect to the two provenance types $a!\kappa_{am} ; \kappa_{ap}$ and $b!\kappa_{bm} ; \kappa_{bp}$. There is a large set of patterns that satisfies this, ranging from the very generic such as the pattern $\mathsf{Any}$ that accepts any provenance types, to the very specific such as $a!\kappa_{am} ; \kappa_{ap} \vee b!\kappa_{bm} ; \kappa_{bp}$ that only accepts the two types $a!\kappa_{am} ; \kappa_{ap}$ and $b!\kappa_{bm} ; \kappa_{bp}$. In general, for a system to be well-typed, all the policies imposed on a channel need to be reconcilable with respect to the provenance of outputs on that channel.

Fundamentally, the type system contains two kinds of types, value types and channel types. Value types are all the types $t$ defined by the syntax of Figure 6.1 and are meant to represent the provenance or approximated provenance of values. Channel types are associated with channels and specify the type of data that may be transmitted on the channel. They represent the provenance policies of principals and therefore specify, based on provenance, what data a channel is allowed to carry. For a system to be well-typed, all values transmitted on a channel need to have a type that is consistent with that of the channel. In typing terms, the channel type of a channel is derived as the intersection of all the types specified by principals for the channel. The type of any value sent on the channel needs to be a subtype of that given by the channel type.

## 6.3 Eliminating provenance tracking

The type and effect system presented in the previous section guarantees that principals always receive data with provenance that matches their policies. As we saw, this means the run time provenance checks become unnecessary. Provenance tracking was maintained in the calculus however as it was needed to prove Subject Reduction. Since the calculus did not provide any means for principals to use the provenance annotations of values, it should be obvious that these and provenance tracking are redundant. We prove this more formally in this section. This is achieved simply by dropping provenance tracking and showing that the transitions derivable that way only differ in the provenance annotations of values when compared to transitions derivable using the reduction relation of the previous section.

Let $\rightarrow_t$ be the reduction relation defined as $\rightarrow_s$ but with the rules for output and input modified as follows.

TRED SND

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_t m\langle\!\langle p : \kappa_p\rangle\!\rangle}$$

TRED RCV

$$\frac{}{a[m{:}\kappa_m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p : \kappa_p\rangle\!\rangle \rightarrow_t a[P\{^{p:\kappa_p}/_{x}\}]}$$

These differ from their counterparts in the relation $\rightarrow_s$ in that they do not perform provenance tracking. Instead, all values retain their original provenance throughout the evolution of the system. If we assume that all values start with empty provenance as is our convention, then the rules above can be rewritten in the more concise form:

TRED SND

$$\frac{}{a[m\,\langle p\rangle] \rightarrow_t m\langle\!\langle p\rangle\!\rangle}$$

TRED RCV

$$\frac{}{a[m\,(\pi\ \mathsf{as}\ x).P] \mid m\langle\!\langle p\rangle\!\rangle \rightarrow_t a[P\{^{p}/_{x}\}]}$$

This makes it clearer that we are indeed dropping explicit provenance annotations, dynamic provenance tracking and dynamic provenance checking from the calculus. Theorem 6.19 below states that systems have the same transitions under this new relation as they do under the relation $\rightarrow_s$. This implies that in well-typed systems, both provenance tracking and checking are redundant.

**Theorem 6.19.** *$S \rightarrow_s S'$ implies that $S \rightarrow_t S''$ and $|S'| = |S''|$. Vice versa, $S \rightarrow_t S'$ implies that $S \rightarrow_s S''$ and $|S'| = |S''|$.*

*Proof.* We need to prove each implication separately. For each one, we assume the antecedent and then proceed by induction on the last rule used in its derivation. In each case, we find that there is a corresponding rule to apply to obtain the consequent. As can seen from the output and input rules of $\rightarrow_s$ and $\rightarrow_t$, the only difference between $S'$ and $S''$ is in the provenance of the values, whereby they are updated in the reduction relation $\rightarrow_s$ but not in the relation $\rightarrow_t$. By erasing the provenance from both resulting systems as expressed by $|S'|$ and $|S''|$, we find that the systems we obtain are indeed the same. To illustrate, let us consider the case when $S \rightarrow_s S'$ is derived using the rule SRed Snd. This means that the transition $S \rightarrow_s S'$ is in fact of the following form:

$$\frac{}{a[m{:}\kappa_m \langle v : \kappa_v \rangle] \rightarrow_s m\langle\!\langle v : a!\kappa_m ; \kappa_v \rangle\!\rangle}$$

We can use the rule TRed Snd to derive the following transition:

$$\frac{}{a[m{:}\kappa_m \langle v : \kappa_v \rangle] \rightarrow_t m\langle\!\langle v : \kappa_v \rangle\!\rangle}$$

which means that in this case $S'' = m\langle\!\langle v{:}\kappa_v \rangle\!\rangle$. We have that $|S'| = |m\langle\!\langle v{:}a!\kappa_m;\kappa_v \rangle\!\rangle| = m\langle\!\langle v \rangle\!\rangle$ and $|S''| = |m\langle\!\langle v : \kappa_v \rangle\!\rangle| = m\langle\!\langle v \rangle\!\rangle$. Hence, we conclude that $|S'| = |S''|$. The other cases follow a similar reasoning. □

## 6.4 Adapting the type system to the full calculus

The type and effect system presented in this chapter works on the subset of the calculus where input-guarded choice, $\Sigma_{i \in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i$, is restricted to indexing sets $I$ of size 0 and 1. As we saw, this is equivalent to replacing input-guarded choice with the nil process 0 and the standard input construct $m{:}\kappa_m\,(\pi \text{ as } x).P$. The full calculus admits arbitrary indexing sets however, with each index corresponding to a different pattern and a different continuation process. This reflects the willingness of principals to receive data with provenance that matches *any* of the patterns specified in the choice construct. There are a number of approaches to adapt the static type and effect system of this chapter to the full calculus. To understand them, let us first start by looking at what input-guarded choice represents from a typing point of view. A choice $\Sigma_{i \in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i$ says that channel $m$ could accept data with provenance that matches any of the types $\pi_i$ and that variable $x$ could be assigned any of these types. This is equivalent to saying that $m$ has the union constraint $\mathsf{c}(\bigcup_{i\in I} \pi_i)$ and that $x$ has the union type $\bigcup_{i\in I} \pi_i$ as we already mentioned. The type system could be easily extended to incorporate this. However, we need to ensure the semantics of input-guarded choice is preserved by the static type system. This means that each continuation process $P_i$ needs to be activated in response to receiving data with provenance that matches the pattern $\pi_i$. This requires a check at run time and defeats the point of the static type system. Instead, what we could do is look at the choice operator as corresponding to an intersection type. That is, in the process $\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i$, we take variable $x$ to have the type $\bigcap_{i\in I} \pi_i$ and channel $m$ to have the constraint $\mathsf{c}(\bigcap_{i\in I} \pi_i)$. Outputs would then be required to satisfy constraints of this form. When all such constraints are satisfied, the system is declared well-typed. At run time, no provenance check is required and a continuation $P_i$ is chosen non-deterministically. This would be faithful to the dynamic semantics of the calculus. The reason is that since all outputs satisfy the intersection type $\bigcap_{i\in I} \pi_i$, then it follows that they also satisfy every pattern $\pi_i$ and therefore the non-deterministic choice is what would have happened in the dynamic semantics. The downside to this approach is that the more patterns $\pi_i$ we have in a choice construct, the stricter the intersection type would be and therefore the more likely it is that it would not be satisfied by the provenance of output data. To get around this, the type system could instead look for a subset of the patterns $\pi_i$ that is satisfied by all outputs on the channel. That is, attempt to satisfy the constraint $\mathsf{c}(\bigcap_{j\in J} \pi_j)$ where $J$ is a subset of $I$. Any non-satisfiable patterns are garbage collected together with their continuation as "unreachable code" by the static analysis. If every input is left with a non-empty set $J$, then the system is declared well-typed. Otherwise, if at least one input is left with an empty choice, then the system would be considered as having failed to type check. To illustrate this, consider the example system below:

$$a[m\langle p\rangle] \mid b[m\langle q\rangle] \mid c[\Sigma_{i\in\{1,2,3\}} m\,(\pi_i \text{ as } x).P_i]$$

where the patterns $\pi_i$ are defined as follows:

$$\pi_1 \triangleq \{a,b\}!\epsilon \qquad \pi_2 \triangleq \mathsf{Any} \qquad \pi_3 \triangleq \{d\}!\epsilon$$

Only patterns $\pi_1$ and $\pi_2$ would be satisfied by both outputs from principals $a$ and $b$. Pattern $\pi_3$ would not be satisfied by data from either principal. As such, the static analysis would assign $x$ the type $\{a,b\}!\epsilon \wedge \mathsf{Any}$ and $m$ the constraint $\mathsf{c}(\{a,b\}!\epsilon \wedge \mathsf{Any})$. The pattern $\pi_3$ and process $P_3$ would be considered unreachable code and pruned from the summation, yielding the optimised system:

$$a[m\,\langle p\rangle] \mid b[m\,\langle q\rangle] \mid c[\Sigma_{i\in\{1,2\}} m\,(\pi_i \text{ as } x).P_i]$$

This system would be declared well-typed by the static type system. At run time, a non-deterministic choice is made between processes $P_1$ and $P_2$, which agrees with the dynamic semantics of the choice operator as the provenance of data sent by $a$ and $b$ would match both $\pi_1$ and $\pi_2$, but not $\pi_3$. Removing unsatisfiable summands could be seen as a static optimisation that removes unreachable code since even with dynamic provenance tracking and checking, such summands would never be satisfied and executed.

## 6.5 Concluding remarks

This chapter presented a type and effect system for our provenance calculus. The primary aim of this type and effect system was to alleviate the performance overhead of provenance tracking and checking by giving static guarantees to enforce the provenance policies of principals. To ensure systems are well-typed under the type and effect system, the policies used by different principals for the same channel have to be consistent, imposing similar restrictions on the provenance of values that may be communicated on that channel. A system where principals specify conflicting policies would not be well-typed under our type system, nor under any static analysis, as such policies are inherently irreconcilable. Nonetheless, this still leaves a large set of useful systems for which the static type system would work. In fact, any system with conflicting policies could be rewritten to avoid such conflicts by associating different policies with different channels.

The type and effect system is interesting in its own right however. Treating provenance as types raises several interesting points when compared to more traditional type systems. The most obvious one as we saw is the fact that provenance is dynamic, leading to a type system where types have a structure resembling that of processes and which changes in accordance with the behaviour of the system. In this sense, they are akin to behavioural types such as those studied by Igarashi and Kobayashi [38] and Acciai and Boreale [2]. Our type and effect system differs in some crucial ways however. Firstly, types are not declared. What we mean by this is that in our calculus values such as channel names are declared without assigning a type to them. Contrast this with most traditional type systems for the $\pi$-calculus where when a channel is declared, via the restriction operator, it is assigned a type. This would usually have the form $(\nu n : t)S$ where channel $n$ is declared to have type $t$ with scope $S$. So how are types obtained in our type system? As we have already seen, we can distinguish between two different types in the type system. The first consists of the provenance annotations of values. These are not constant but change as the system evolves. They are under the control of the run time environment and are not assigned to values by the code. The second kind is found in the provenance policies associated with channels. Provenance policies, seen as types, are similar to the channel types found in other type systems for the $\pi$-calculus in that they specify what data the channel is allowed to carry. Each use of a channel name for input specifies a (possibly different) type for it. As a result, it is possible for a channel name to be associated with conflicting provenance policies and hence have a non-satisfiable type. This actually highlights another difference, that of the scope of a type. In type declarations of the form $(\nu n : t)S$, all occurrences of the channel name $n$ in $S$ have the same fixed type $t$. On the other hand, in our type system every instance of a value will have its own type, whether in the form of a provenance annotation or a provenance policy. Our type system acts as a type inference system, attempting to approximate the provenance of values and unify the different provenance policies associated with a channel. Systems that fail to be type checked are those where the unified type for the provenance policies can not be satisfied by the approximated provenance of the values.

We end this chapter by discussing how the type system could be implemented in a distributed setting. The key to this is in the compositionality of the type system. We already mentioned that compositionality is required in proofs of typability as it enables us to type check the subsystems of a given system separately and then join them to get a type derivation for the whole system. Not only that, but compositionality is also very important in implementations of the type system to ensure that disparate systems can be type checked in isolation. This is done by type checking a given system with respect to the *type interfaces* of other systems it is expected to interact with. The type interfaces of systems are given by constraint environments in the case of our type system. To illustrate this, recall the typing rule for parallel composition:

$$\frac{\Theta\,,\Xi_2 \circ \Gamma \vdash S_1 \triangleright \Xi_1 \quad \Theta\,,\Xi_1 \circ \Gamma \vdash S_2 \triangleright \Xi_2}{\Theta \circ \Gamma \vdash S_1 \mid S_2 \triangleright \Xi_1\,,\Xi_2}\; av(\Xi_1) \cap av(\Xi_2) = \emptyset$$

The constraint environments $\Xi_1$ and $\Xi_2$ give the typing interfaces of systems $S_1$ and $S_2$ respectively. Each system only needs to be type checked with respect to the constraint environment of the other system; there is no need to know the internal structure of the other system. This is what is expressed by the two premises $\Theta\,,\Xi_2 \circ \Gamma \vdash S_1 \triangleright \Xi_1$ and $\Theta\,,\Xi_1 \circ \Gamma \vdash S_2 \triangleright \Xi_2$. Constraint environments are similar in purpose to interfaces in programming languages such as Java and header files in programming languages such as C. They allow one to extract the public interface of a system in order to verify that other systems are compatible with this system and therefore are able to interact with it in a safe manner.

# Chapter 7

# Security of Provenance

In this chapter, we introduce an extension to the calculus that aims to give principals some control over the disclosure of provenance. The main motivation behind this is security. When a provenance annotated value is published by a principal, its provenance reveals the identity of this principal to any other principal that may consume the value. Not only that, but it also reveals the identities of other principals further back in the chain of custody of the value and of any channels it was communicated on. This type of information may be sensitive and its disclosure may need to be tightly controlled. Naturally, this would conflict with the ability of other principals to make fully informed decisions about the data they consume. Furthermore, the *provenance disclosure policy* of one principal may conflict with that of another principal in the chain of custody of the value. All these factors make this an interesting and non-trivial problem to study.

After giving the motivation for this chapter, we take a step back and look at the more general problem of allowing principals to control the provenance tracking process. We consider the possible effects on provenance that such a change would have. The answer we arrive at is that principals should only be allowed to decrease the provenance of a value, never to increase it. We formalise this by introducing the concept of *weakenings*. To weaken a provenance sequence means simply to reduce the amount of information that it contains about the past. For example, the pattern $\sim!\epsilon\ ;\ \epsilon$ can be interpreted to tell us that *someone* sent an original value on a channel with empty provenance. This can be considered a weakening of the provenance sequence $a!\epsilon\ ;\ \epsilon$ which tells us a little more than the pattern by revealing the exact identity of the principal that sent the value. Weakening allows principals to control what parts of a provenance sequence are disclosed to other principals, and this gives them the ability to implement their provenance disclosure policies. We incorporate weakenings into the calculus by introducing another concept, that of *filters*. Filters associate different weakenings of a provenance sequence with different principals, thus allowing a principal to selectively disclose provenance information. We keep filters abstract to ensure our results remain general, but we do demonstrate their applicability by proposing a sample filter language and giving a wide range of examples. As with the original calculus described in Chapter 3, we assume that the provenance tracking infrastructure is trusted and therefore any provenance it records is trusted. Furthermore, we assume it is also secure, guaranteeing that provenance cannot be forged and filters cannot be circumvented by principals. Therefore, principals are only able to access provenance that has not been masked from them by filters.

The remainder of this chapter is split into four main sections. Section 7.1 looks at weakenings, Section 7.2 extends the calculus with filters while Section 7.3 proposes a sample filter language and demonstrates its use through several examples. Section 7.4 concludes the chapter.

## 7.1 Controlling the disclosure of provenance

### 7.1.1 Motivation

Tracking provenance in the calculus, as presented in Chapter 3, is performed automatically as part of the provenance tracking reduction relation. This, in effect, corresponds to a model where provenance tracking is handled *entirely* by the run time environment. Principals are only given the ability to make boolean tests against the provenance information, with no control at all over how this information is gathered or who may use it. The provenance of data, however, affects the behaviour of principals both directly and indirectly. It affects the behaviour of a principal directly as it is used by the principal to determine what data to receive and what branch of computation to proceed with. It affects it indirectly as it is also used by other principals to make their own decisions as to which data to receive and which computation branch to take, thus determining, among other things, the possible interactions they could engage in with the principal. This motivates the need to give principals some control over the provenance information and the provenance tracking process.

More specifically, however, two arguments may be raised in favour of giving principals control over provenance tracking.

1. **Security.** The first argument concerns access control and aims to address privacy and security issues arising from provenance disclosure. The provenance of a value

records its entire history and reveals, to *any* principal that gets hold of the value, the identity of every principal involved in getting it to that state. This may, however, conflict with the security policies of some principals. We would expect the security policy of a principal to restrict access to sensitive provenance information and only give other principals access to nonsensitive provenance information. More generally, the privileges of each principal will determine how much provenance they are entitled to access.

2. **Optimization.** The second argument pertains to the size and level of detail of a provenance sequence. As provenance tracking strictly appends new events to a provenance sequence, this latter will only expand with time and, in the presence of "loops" for example, will lead to long and repetitive provenance sequences. This calls for a mechanism that prunes irrelevant events from provenance sequences and allows us to abstract away from unimportant details in the provenance of data.

We tackle this problem by breaking it into two parts. We start by considering *what* aspects of provenance tracking principals should be involved in. Then, based on this, we look at *how* to extend the calculus with programming constructs that allow us to implement such features. Throughout, our guiding principle will be to preserve the *integrity* of the provenance information. By this, we mean that provenance should always remain correct with respect to the past of the system. We have already defined what it means for provenance to be correct in Section 5.3. To recap, recall that we interpret the meaning of provenance as one or more logical statements about the past of the system. If the truth of these logical statements follows from the past of the system, then we say that the provenance is correct. Intuitively, this means that what the provenance tells us about the past of the system did in fact take place. Therefore, to preserve the integrity of provenance means to disallow any modifications (or in other words forgeries) to provenance that would result in claims contrary to what actually took place. We make these ideas more formal by defining the notions of weakening and filter language in the following section.

The approach we take assumes that the run time infrastructure responsible for tracking provenance is also responsible for its security. This is reflected in the extension of the calculus presented in this chapter. In this extended calculus, principals are able to specify their security requirements in the form of *filters*. Note that filters act only as *specifications* of what parts of provenance principals wish to restrict from other principals. Enforcement of these filters rests with the run time infrastructure however, which as defined by the operational semantics of the calculus, ensures that principals can only access provenance that has not been hidden from them by filters. We assume this cannot be circumvented by principals.

### 7.1.2 Weakening of provenance sequences

First, let us consider what *effects* on provenance the involvement of principals in provenance tracking should have. This will make it easier to determine the acceptable ways in which principals may control provenance tracking. As we explained earlier, we view the provenance of a value as essentially a set of assertions about the past of that value. Hence, any involvement of principals in provenance tracking would have the effect of changing this set of assertions, either by adding or removing some assertions. However, in order to have trust in the provenance information, it is important that this information is correct. This means that even if principals are involved in provenance tracking, this involvement should not result in false provenance assertions. By this reasoning, any addition of false assertions to the provenance information is to be rejected. The addition of true assertions would be acceptable, but it would result in provenance information that is unnecessary with respect to the particular notion of provenance we are dealing with. So, we are left with the removal of assertions, or in other words the decrease of what a provenance sequence tells us about the past, as the only possible way principals may control provenance. We call this the *weakening* of a provenance sequence.

To determine what weakening corresponds to in concrete terms, let us consider the structure and elements of provenance sequences. A provenance sequence is made up of zero or more ordered events. Each event may be seen as being composed of three parts: 1. the author of the event, 2. the type of the event, and 3. the provenance of the channel used. So, in addition to the temporal ordering of events, this yields four kinds of information that a provenance sequence contains. Removing any of these constitutes a weakening of the provenance sequence. To accomplish this, we propose using the pattern language from Section 3.3.1 as the target for weakening; that is, the weakening of a provenance sequence is going to be a term in the pattern language. Let us now illustrate how patterns implement the four types of weakening. For an event $a!\kappa$, weakening its author may be done using any pattern of the form $G!\kappa$ where $a \in [\![G]\!]$. Weakening its type may be accomplished by the pattern $a!\kappa \vee a?\kappa$ while weakening the provenance of its channel may be accomplished using any event $a!\pi$ where $\pi$ is a weakening of $\kappa$. For concatenation, for example in the provenance sequence $\kappa; \kappa'$, the pattern $(\kappa; \kappa') \vee (\kappa'; \kappa)$ may be considered a weakening for it since we cannot tell from the latter which of the two subsequences $\kappa$ or $\kappa'$ took place first.

It should be clear from the examples above that for a given provenance sequence $\kappa$, any pattern $\pi$ such that $\kappa \models \pi$ may be considered a weakening for it. Looking at this from a typing point of view, the pattern $\pi$ is simply any valid type for $\kappa$, including $\kappa$ itself as the most specific type (strongest weakening) and any supertype of it as a possible (weaker) weakening. This yields a very flexible weakening notion allowing a provenance sequence to be weakened by different "amounts" depending on the requirements of the particular provenance disclosure policy. We define weakening of provenance sequences and its generalisation to patterns in definitions 7.1 and 7.2 respectively.

**Definition 7.1** (Weakening of provenance sequences)**.** We say that a provenance sequence $\kappa$ is weakened by a pattern $\pi$, written $\kappa \leq \pi$, when it holds that $\kappa \models \pi$.

**Definition 7.2** (Weakening of patterns)**.** We extend the notion of weakening to patterns as follows. We say that a pattern $\pi$ is weakened by a pattern $\pi'$, written $\pi \leq \pi'$, if and only if for every provenance sequence $\kappa$, it holds that if $\kappa \models \pi$ then $\kappa \models \pi'$.

Note that the definition of weakening of patterns can be considered as an extension to that of provenance sequences since the set of provenance sequences is a subset of the set of patterns. Note also that the weakening relation is a partial order as shown below.

**Proposition 7.3.** *The relation $\leq$ is a partial order.*

*Proof.* By showing that $\leq$ is reflexive, antisymmetric and transitive. The details are straightforward and are hence omitted. □

Weakening does indeed meet our requirements and is enough for us to implement both kinds of provenance control we are interested in. Abstraction maps directly to weakening since a provenance sequence can be replaced with a pattern to hide irrelevant provenance information. For example, $e; \mathsf{Any}$ hides the details of the tail of the provenance sequence $e; \kappa$, while $\kappa*$ abstracts away from the exact number of times $\kappa$ took place. For access control, what we will need to do is assign different weakenings to different principals based on their privileges. In the following section, we show how the calculus can be extended with programming constructs to accomplish this.

## 7.2 Filters and calculus extensions

### 7.2.1 Filter languages

We call a specification for the change of provenance visibility a *filter*. The application of a filter to a provenance sequence is meant to associate with each principal a weakening of this provenance sequence; we refer to this weakening as the principal's *view* of the provenance sequence. We use $\phi, \psi, \ldots$ to range over filters and $\Phi$ for the set of all filters. We denote by $\phi \triangleright \kappa$ the application of filter $\phi$ to provenance sequence $\kappa$. In order to incorporate this into the calculus, we add the clause $\phi \triangleright \kappa$ to the syntax of provenance sequences given in Figure 3.5. We refer to the set of provenance sequences extended with the above clause as *filtered provenance sequences* and to the subset that contains no filter application as *unfiltered provenance sequences*. We reserve the letter $\mathcal{K}$ for the set of unfiltered provenance sequences and use $\mathcal{K}_\Phi$ to refer to filtered provenance sequences, where the filters are drawn from set $\Phi$. To relate the two, we define the *filter erasure* function $|-| : \mathcal{K}_\Phi \rightarrow \mathcal{K}$ which removes all filters from a provenance sequence. The filter erasure function is defined inductively on the structure of filtered provenance sequences as given in Definition 7.4.

**Definition 7.4.** The filter erasure function $|-| : \mathcal{K}_\Phi \rightarrow \mathcal{K}$ removes all filters from a given provenance sequence. It is defined as follows:

$$|\epsilon| = \epsilon \qquad |e| = e \qquad |\kappa\,;\kappa'| = |\kappa|\,;|\kappa'| \qquad |\phi \triangleright \kappa| = |\kappa|$$

As we mentioned already, filters will be used to give each principal its own view of the provenance sequence in question. In light of this, the semantics of filters would be to associate with each principal and filtered provenance sequence a weakening of that provenance sequence. We formalise this in the definition of *filter languages* given below.

**Definition 7.5.** A filter language is a tuple $(\Phi, view)$ where $\Phi$ is a set of filters (we use the letters $\phi, \psi, \ldots$ to range over filters) and $view : \mathcal{A} \times \mathcal{K}_\Phi \rightarrow \Pi$ is a function such that for all principals $a$, provenance sequences $\kappa$ and patterns $\pi$, it holds that if $view(a, \kappa) = \pi$ then $\pi$ is a weakening of $|\kappa|$.

The above gives us a generic definition of filter languages. Its only requirement is that filters must give each principal a view of the provenance sequence in question which, as explained before, must be a weakening of the provenance sequence. In the following section, we extend the calculus with filters. We opt to take the same abstract approach we took with patterns in Chapter 3 and make the calculus parametric on the choice of a concrete filter language. Like with patterns, this allows us to keep our results generic and applicable to any particular filter language as long as it conforms to the above definition.

### 7.2.2 Syntactic changes

We now look at how the calculus is adapted to use filters. Instead of adding a new primitive to apply a filter to the provenance of a value, we elect instead to modify the output construct so that it takes two parameters: the annotated value to be sent, and a filter that would be applied to the resulting provenance sequence of the value. In addition to keeping the calculus simple, this matches our intuition that principals would usually apply filters to values when they decide to publish them to other principals. It also allows the filter to reference the complete provenance sequence including the additions to be made as a result of the output operation itself. We summarise the syntactic changes in Figure 7.1.

Figure 7.1. *Modified syntax with filters*

| | |
|---|---|
| $P ::=$ | process terms |
| $\quad\ldots$ | $\quad$ all clauses except output unchanged |
| $\quad w\langle w \;\mathsf{at}\; \phi\rangle$ | $\quad$ modified output construct with filters |
| $\kappa ::=$ | provenance sequences |
| $\quad\ldots$ | $\quad$ same as before |
| $\quad\phi \triangleright \kappa$ | $\quad$ filtered sequence |

### 7.2.3 Semantic changes

In addition to the syntactic changes, we also modify the semantics of the calculus to take filters into account. We use $\equiv_f$ and $\rightarrow_f$ to denote the structural congruence and reduction relations of the new calculus respectively. Structural congruence stays the same while only the input and output rules in the reduction relation change.

The new reduction rules are given in Figure 7.2. Rule FRED SND states that when a value $p{:}\kappa_p$ is sent by a principal $a$ on a channel $m{:}\kappa_m$, its provenance is updated as before to be $a!\kappa_m\,;\kappa_p$. In addition to this, the latter is also annotated with the filter $\phi$ specified in the output construct to give us the filtered provenance sequence $\phi\triangleright a!\kappa_m;\kappa_p$. In the FRED

Rcv rule, pattern matching is done against the view of the receiving principal, $a$ in this case, as opposed to the original provenance sequence as done in the rule ARed Rcv. Principal $a$'s view of the provenance sequence $\kappa_p$ is given by the function $view(-)$ as $view(a, \kappa_p)$. The result of the function $view(-)$ is a pattern, and hence the check that we make in the input rule is a pattern weakening check (instead of the satisfaction check we made in the original semantics). If $\pi_j$ (where $j \in I$) is a weakening of $view(a, \kappa_p)$, then $P_j$ is chosen for the continuation and the appropriate substitution is made. As before, if more than one such pattern exists, one of them is chosen nondeterministically.

Figure 7.2. *Modified semantics with filters*

FRed Snd

$$\frac{}{a[m{:}\kappa_m \langle p{:}\kappa_p \;\mathsf{at}\; \phi\rangle] \rightarrow_f m\langle\!\langle p : \phi \triangleright a!\kappa_m \,;\, \kappa_p \rangle\!\rangle}$$

FRed Rcv

$$\frac{view(a, \kappa_p) \leq \pi_j \qquad j \in I}{a[\Sigma_{i \in I} m{:}\kappa_m\,(\pi_i \;\mathsf{as}\; x).P_i] \parallel m\langle\!\langle p : \kappa_p \rangle\!\rangle \rightarrow_f a[P_j\{^{p:a?k_m;\kappa_p}/_x\}]}$$

## 7.3 A sample filter language

In the previous section, we gave a generic definition of the filter language but left its details unspecified. This is useful since it allows us to focus on the more important aspects of filters and weakenings without having to worry about the details of the filter language. It also means that any results we show would be applicable to any setting as long as the concrete filter language chosen complies with Definition 7.5. For the purpose of giving examples, and also to provide a more in-depth analysis of filters and their effects on provenance tracking, we give a sample filter language in this section.

Probably the simplest filter language is the one where each principal name is mapped to the pattern intended to act as its view of the provenance sequence in question. This would look something like:

$$(a_1 \mapsto \pi_1, \ldots, a_n \mapsto \pi_n) \triangleright \kappa$$

where the view of $\kappa$ assigned to $a_i$ is $\pi_i$. The semantics of this filter language can be given by the following simple clause:

$$view(a_i, (a_1 \mapsto \pi_1, \ldots, a_i \mapsto \pi_i, \ldots, a_n \mapsto \pi_n) \triangleright \kappa) = \pi_i$$

Hence, in order for this language to be compliant with the definition of filter languages given in Definition 7.5, all patterns $\pi_i$ need to be weakenings of $|\kappa|$. While such a language could indeed act as a sample pattern language, it is unduly verbose and inflexible. For instance, principals need to explicitly specify the full view of provenance they want to associate with each principal. They also need to make sure the patterns they specify are correct weakenings of the provenance sequence in question. More importantly, the filters specified by a principal override all previous filters applied by other principals, which may violate the provenance disclosure policies of those principals.

The filter language we propose to use instead gives principals full ownership of the provenance of data they publish. It allows them to control the disclosure of provenance pertaining to their own actions, as well as that relating to other principals' actions (subject to the provenance disclosure policies of those principals). This enables principals to change the visibility of provenance sequences in a controlled manner. What we mean by this is that while principals are able to change the views assigned to other principals, they can only do so in accordance with their own privileges. The mechanism for achieving this consists of enabling principals to decide whether the view to be assigned to another principal is computed with respect to their own view (which intuitively corresponds to the principal sharing parts of their view with the other principal) or with respect to the other principal's existing view (in effect weakening the other principal's view even further). This is explained in more detail in Section 7.3.1.

To keep the sample filter language simple, we only target a subset of the pattern language. We let principals hide both single principal names and whole subsequences. We do this by using the following two forms of weakening.

- Hiding principal names: the principal name $a$ in the sequence $a!\kappa;\kappa'$ can be hidden using the pattern $\sim!\kappa\,;\kappa'$.
- Hiding subsequences: the provenance sequence $\kappa$ in $a!\kappa\,;\kappa'$ can be hidden using the pattern $a!\mathsf{Any}\,;\kappa'$.

What this means is that weakenings can be formally defined as the subset of patterns given by the following grammar.

Figure 7.3. *The filter language: weakenings*

| | | |
|---|---|---|
| $\pi$ ::= | | weakening |
| | $\epsilon$ | empty sequence |
| | $\alpha$ | single event |
| | $\pi ; \pi$ | concatenation |
| | Any | hidden sequence |

| | | |
|---|---|---|
| $\alpha$ ::= | | event |
| | $G!\pi ; \pi$ | send event |
| | $G?\pi ; \pi$ | receive event |
| $G$ ::= | | group |
| | $a$ | principal name |
| | $\sim$ | hidden principal |

For the remainder of this section, when we refer to weakenings it should be understood to mean this restricted subset of patterns, unless otherwise stated. Note that the patterns used in the input construct are drawn from whatever pattern language the calculus is instantiated with, and hence from the full set of patterns when that pattern language is the sample language given in Chapter 3.

## 7.3.1 Syntax

### 7.3.1.1 Filters, mappings and modes

The syntax of the filter language is given in Figure 7.4. A filter $\phi$ is a term of the form $a.F$ where $a$ denotes the principal that issued the filter and $F$ is a filter mapping. Filter mappings simply associate with each group expression $G$ a mode $m$ and a filtering expression $E$. We denote each such association by $G \mapsto_m E$ and use $\varnothing$ for the empty mapping and $F; F$ for the composition of mappings. Group expressions are the same as in Section 3.3.1 and enable us to associate a filtering expression with more than one principal. Modes affect the way filter expressions are applied and may be one of two kinds: casc, which stands for cascade; and over, which stands for override.

To understand the role of modes, consider that a particular provenance sequence will get tagged with a filter each time it is published by a principal. This results in provenance sequences with multiple nested filters such as $\phi \triangleright \kappa; \phi' \triangleright \kappa'$ for example, where $\phi'$ was applied to $\kappa'$ after which the events in $\kappa$ took place and then $\phi$ was applied to the resulting sequence. Now, if we consider how the view of some principal, say $a$ for example, may be computed, we find two natural methods arise: either with respect to $a$'s view of $\kappa; \phi' \triangleright \kappa'$, which we refer to as casc mode; or with respect to the issuer's view of $\kappa; \phi' \triangleright \kappa'$, which we refer to as over mode. The mode casc only permits the views that principals already have of a provenance sequence to be weakened. On the other hand, the mode over allows both weakening and "strengthening", leading to a mechanism which enables full dynamic change of provenance visibility. Note, however, that a principal's view can only be strengthened up to that of the issuer; this is what we referred to previously when we said that our sample filter language allows principals to change the visibility of provenance in a *controlled manner*. Note also that the name of the principal that issued the filter is kept for the purpose of calculating views when the mode specified is over.

Figure 7.4. *The filter language: syntax*

| | | | |
|---|---|---|---|
| $\phi$ ::= | filters | $E$ ::= | filtering expressions |
| $a.F$ | issuer $a$ and mapping $F$ | $H$ | selector |
| | | $H/E$ | navigation |
| $F$ ::= | filter mappings | $[\pi]E$ | conditional |
| $\varnothing$ | empty mapping | $E \circ E$ | composition |
| $G \mapsto_m E$ | singleton mapping | $A$ | filter name |
| $F;F$ | composition | | |
| | | $H$ ::= | selectors |
| | | *all* | select entire subtree |
| $m$ ::= | modes | *author* | select author only |
| casc | cascade | *channel* | select channel only |
| over | override | *tail* | select tail only |

### 7.3.1.2 Filter expressions

Filter expressions specify the parts of a provenance sequence that need to be hidden or masked from principals. The parts of a provenance sequence which are not selected by the filter are what constitutes the principal's view. We can also envisage a complementary setup where filter expressions specify what is visible and everything else is made hidden from principals. The choice between these two is mostly a matter of taste. Alternatively, we can even have a filter language that combines both of these types of filter expressions, allowing principals to choose to either specify what provenance to hide (negative filters) or what provenance to show (positive filters). We feel negative filters are expressive enough to allow us to encode all provenance disclosure policies we are interested in and hence find no real need to include positive filters as well.

Filter expressions treat provenance sequences as nested lists, each of which is composed of a head and a tail. They allow principals to manipulate the top-most level of the provenance sequence as well as delve deeper into its subsequences. The simplest class of filter expressions is that of selectors $H$, which work at the top-most level of the provenance sequence. They are meant to reflect the structure of the provenance sequence and allow principals to reference its different parts. There are four types of selectors. The selector *all* hides the entire provenance sequence in the current context, which means it would transform any sequence $\kappa$ to $\mathsf{Any}$. The selector *author* hides the author of the most recent event in the provenance sequence. So, for example, applying *author* to the provenance sequence $a!\kappa\,;\kappa'$ would result in $\sim!\kappa\,;\kappa'$. The third selector is *channel* and it hides the channel of the most recent event, transforming the sequence $a!\kappa;\kappa'$ to $a!\mathsf{Any}\,;\kappa$. The last selector, *tail*, hides the tail of the provenance sequence. For example, applying it to $a!\kappa\,;\kappa$ would result in $a!\kappa\,;\mathsf{Any}$.

Seen as a nested list, a provenance sequence such as $a!\kappa\,;\kappa'$ contains two subsequences, the channel, $\kappa$, and the tail, $\kappa'$. To access these, principals may use the two filter expressions $channel/E$ and $tail/E$. These apply the filter expression $E$ to the channel provenance, $\kappa$, and tail provenance, $\kappa'$, respectively. The conditional filter expression, $[\pi]E$, applies the filter expression to the provenance sequence if it matches the pattern $\pi$ and leaves it unaltered otherwise. The form $E \circ E'$ denotes the composition of filters $E$ and $E'$. The result of applying it to a provenance sequence is that of applying $E$ to the sequence obtained as a result of applying $E'$. Filter names $A$ allow us to express recursive filters and are necessary to be able to express some of our examples as we will see later.

### 7.3.2 Semantics

The structure of filters reflects their expected role as discussed earlier, where we have a component that is responsible for weakening provenance sequences (filtering expressions, modes and the name of the issuer), and another for associating weakenings with principals (filter mappings). When the aim of the filter is to abstract away from unimportant information, we expect all principals to be mapped to the same view, whereas in the case of security policies we would likely find different principals given different views depending on the requirements of the particular provenance disclosure policy.

We view filters as mappings from principals to modes and filtering expressions, and hence define the notions of *domain* of a filter and *image* of a principal under a filter below. The domain of a filter gives the set of groups which are assigned some mode and filtering expression by the filter, and is intended to account for those principals explicitly addressed by the filter. The image of a principal under a filter is a pair composed of the mode and filtering expression associated with the principal and is inferred from their group membership.

**Definition 7.6.** The issuer of a filter $\phi = a.F$ is the principal $a$.

**Definition 7.7.** The domain of a filter mapping $F$, written $dom(F)$, is defined by induction on the structure of $F$ as follows:

$$dom(\varnothing) = \emptyset \qquad dom(G \mapsto_m F) = \{G\} \qquad dom(F; F') = dom(F) \cup dom(F')$$

The domain $dom(\phi)$ of a filter $\phi = a.F$ is the domain of its mapping $F$.

**Definition 7.8.** The image of a principal $a$ under a filter mapping $F$, written $img(a, F)$, is defined as follows:

$$img(a, \varnothing) = \emptyset \qquad img(a, F; F') = img(a, F) \cup img(a, F')$$

$$img(a, G \mapsto_m E) = \begin{cases} \{(m, E)\} & \text{if } a \in [\![G]\!] \\ \emptyset & \text{otherwise} \end{cases}$$

The image of a principal $a$ under a filter $\phi = a.F$ is the image of the principal under the mapping $F$.

The semantics of filters is given by defining the function *view*, which returns for every principal and filtered provenance sequence a pattern representing that principal's view of the provenance sequence. The function *view* is only defined when all filters in the provenance sequence are both *exhaustive* and *non-ambiguous*. These two properties ensure that every principal gets exactly one view and are defined in 7.9 and 7.10 respectively. An alternative to requiring filters to be exhaustive and non-ambiguous would be to assign a default filtering expression to every principal not included in the filter and to assume a first-match (or last-match) policy whereby a principal is assigned the first (or last) filter mapping that applies to them.

**Definition 7.9** (Exhaustivenss)**.** A filter $\phi$ is exhaustive if every principal belongs to *at least* one group in the domain of the filter, that is $\bigcup_{G \in dom(\phi)} [\![G]\!] = \mathcal{A}$.

**Definition 7.10** (Non-ambiguity)**.** A filter $\phi$ is non-ambiguous if every principal belongs to *at most* one group in the domain of the filter, that is $[\![G]\!] \cap [\![G']\!] = \emptyset$ for all distinct groups $G$ and $G'$ in $dom(\phi)$.

It is easy to check that an exhaustive, non-ambiguous filter assigns every principal exactly one image. To aid in the definition of the semantics of the filter language, we define the projection of a filter $\phi$ onto a principal $a$, which we denote by $\phi_a$, to be the tuple containing the principal that issued the filter, the mode and the filtering expression assigned to the principal. This tuple summarises all the information in a filter that is relevant to a particular principal and is needed to compute their view of a provenance sequence.

**Definition 7.11.** The projection $\phi_a$ of an exhaustive, non-ambiguous filter $\phi$ onto a principal $a$ is the tuple $(b, m, E)$ where $b$ is the issuer of $\phi$ and $(m, E)$ is the image of $a$ under $\phi$.

We give the formal semantics of filters in Figure 7.5. As in Section 5.3, it turns out to be simpler to use a modified grammar for provenance sequences. The two grammars are equivalent for our purposes however, and any choice of one over the other is merely for convenience.

Figure 7.5. *The filter language: semantics of filter mappings*

$$\mathit{view}(a, \epsilon) \triangleq \epsilon$$
$$\mathit{view}(a, e\,;\kappa) \triangleq e\,;\mathit{view}(a, \kappa)$$
$$\mathit{view}(a, \phi \triangleright \kappa) \triangleq \begin{cases} \mathit{apply}(E, \mathit{view}(a, \kappa)) & \text{if } \phi_a = (b, \mathsf{casc}, E) \\ \mathit{apply}(E, \mathit{view}(b, \kappa)) & \text{if } \phi_a = (b, \mathsf{over}, E) \end{cases}$$

A principal's view of the empty provenance sequence $\epsilon$ is simply the empty sequence itself as there are no filters in this case. For concatenation $e\,;\kappa$, the view of principal $a$ is given as $e\,;\mathit{view}(a, \kappa)$, where the event $e$ is left unaltered as no filter is applied to it, and is concatenated to $a$'s view of the remainder of the sequence $\kappa$. For a filtered provenance sequence $b.F \triangleright \kappa$, the view of principal $a$ depends on the mode specified: if the mode is casc then it is given as $\mathit{apply}(E, \mathit{view}(a, \kappa))$, that is by applying the filter expression $E$ associated with $a$ to their view of the rest of the sequence $\kappa$; otherwise if the mode is over, $a$'s view is given as $\mathit{apply}(E, \mathit{view}(b, \kappa))$, which is to say by applying $E$ to the issuer's own view. The function *apply* takes a filter expression and a pattern and returns a weakening of that pattern. It gives the semantics of filter expressions and its definition is given in Figure 7.6.

We have already explained informally how each of the different filter expressions work. It is probably worth iterating that filter expressions are used to select parts of a provenance sequence, and that those parts that are selected are hidden from principals by replacing them with the pattern $\sim$ or Any as applicable. The parts of the provenance sequence not selected are left visible to the principal. The evaluation of filter names is done with respect to a global set of definitions $\mathcal{D}_\Phi$, which associates with each filter name $A$ a filtering expression $E$. Each such association is denoted by:

$$\mathsf{fil}\ A = E$$

The filter expression $E$ associated with $A$ is given by $\mathcal{D}_\Phi(A)$. To guarantee that filter names are well defined, we impose the same well-formedness condition on the set $\mathcal{D}_\Phi$ as the one imposed on pattern name definitions.

Figure 7.6. *The filter language: semantics of filter expressions*

$$apply(H,\pi) \triangleq \begin{cases} \mathsf{Any} & \text{if } H = all \\ \sim!\pi' \,;\pi'' & \text{if } H = author \text{ and } \pi = G!\pi' \,;\pi'' \\ G!\mathsf{Any} \,;\pi'' & \text{if } H = channel \text{ and } \pi = G!\pi' \,;\pi'' \\ G!\pi' \,;\mathsf{Any} & \text{if } H = tail \text{ and } \pi = G!\pi' \,;\pi'' \\ \pi & \text{otherwise} \end{cases}$$

$$apply(H/E,\pi) \triangleq \begin{cases} G!apply(E,\pi') \,;\pi'' & \text{if } H = channel \text{ and } \pi = G!\pi' \,;\pi'' \\ G!\pi' \,;apply(E,\pi'') & \text{if } H = tail \text{ and } \kappa = G!\pi' \,;\pi'' \\ \pi & \text{otherwise} \end{cases}$$

$$apply([\pi']E,\pi) \triangleq \begin{cases} apply(E,\pi) & \text{if } \pi \leq \pi' \\ \pi & \text{otherwise} \end{cases}$$

$$apply(E \circ E',\pi) \triangleq apply(E, apply(E',\pi))$$

$$apply(A,\pi) \triangleq apply(E,\pi) \quad \text{where } \mathcal{D}_\Phi(A) = E$$

*Note: All rules for send events $a!\pi$ are similarly applicable to receive events $a?\pi$.*

Our sample filter language guarantees that the views granted to principals are always weakenings of the underlying provenance sequence. This is proved formally in Proposition 7.14. The following two lemmas are needed for the proof of the proposition.

**Lemma 7.12.** *$\pi \leq \pi'$ implies that $\alpha \,;\pi \leq \alpha \,;\pi'$.*

*Proof.* Assume it is the case that $\pi \leq \pi'$. Also assume that there exists a provenance sequence $\kappa$ such that $\kappa \models \alpha \,;\pi$. It is easy to check that, from the definition of pattern satisfaction, $\kappa$ must be of the form $e \,;\kappa'$ where $e \models \alpha$ and $\kappa' \models \pi$. Recall that, from the definition pattern weakening, since $\kappa' \models \pi$ then it must be that $\kappa' \models \pi'$. From this latter, we can infer that $e\,;\kappa' \models \alpha\,;\pi'$. Hence, we conclude that $\alpha\,;\pi \leq \alpha\,;\pi'$. □

**Lemma 7.13.** *$\pi \leq \pi'$ implies that $\pi \leq apply(E, \pi')$.*

*Proof.* The full proof of this proceeds by induction on the definition $apply(-)$, however it is be enough to observe that the result of $apply(-)$ is always either the same pattern we started with or a weakening of it. □

**Proposition 7.14.** *The language defined in this section is a filter language.*

*Proof.* We need to show that it is always the case that $view(a, \kappa)$ is a weakening of $|\kappa|$, that is, $|\kappa| \leq view(a, \kappa)$. This can be done by induction on the definition of the function $view(-)$. There are three cases.

In the first case, we have $view(a, \epsilon) = \epsilon$. So the properly clearly holds since $\epsilon \leq \epsilon$.

In the second case, we have $view(a, e;\kappa') = e;view(a, \kappa')$. From the induction hypothesis, we can get $|\kappa'| \leq view(a, \kappa')$, and from this latter, we can conclude by Lemma 7.12 that $|e\,;\kappa'| \leq e\,;view(a, \kappa')$.

In the last case, we have $view(a, \phi \triangleright \kappa') = apply(E, view(a, \kappa'))$ if $\phi_a = (b, \mathsf{casc}, E)$ or $view(a, \phi \triangleright \kappa') = apply(E, view(b, \kappa'))$ if $\phi_a = (b, \mathsf{over}, E)$. The induction hypothesis gives us $|\kappa'| \leq view(a, \kappa')$ and $|\kappa'| \leq view(b, \kappa')$ respectively. So all we need to do is show that $|\kappa'| \leq view(a, \kappa')$ and $|\kappa'| \leq view(b, \kappa')$ imply $|\kappa'| \leq apply(E, view(a, \kappa'))$ and $|\kappa'| \leq apply(E, view(b, \kappa'))$, which follow directly from Lemma 7.13 □

## 7.3.3 Examples

In this section, we give a few example filters and systems to illustrate the use of our sample filter language and the interplay between filters and provenance tracking in the calculus. We start by defining some useful filters and then use them to define systems that model some common scenarios and settings.

#### 7.3.3.1 Useful filters

Here we only define the filter expressions. We can then plug them into the appropriate filter mappings depending on the requirements of the system we are interested in implementing.

**Turning off provenance tracking.** The simplest filter expression is the one that hides all provenance information. We denote it by $\mathit{off}_\Phi$ and define it as follows:

$$\mathit{off}_\Phi = \mathit{all}$$

This filter expression hides all provenance, whether relating to the principal's own actions or to those of other principals. This is consistent with the aims of filters as we want principals to be able to hide their own actions as well as those of their "sources". If every principal were to apply this filter to all their outputs, it would have the effect of turning off provenance tracking completely.

**Identity filter.** The filter expression $\mathit{off}_\Phi$ hides all provenance information; at the other end of the spectrum we find the identity filter expression $\mathit{id}_\Phi$. This latter does not hide any provenance from principals. It simply keeps the view of provenance given to principals unchanged. The $\mathit{id}_\Phi$ filter is defined as follows:

$$\mathit{id}_\Phi = [\mathbf{F}]\mathit{all}$$

where **F** stands for any unsatisfiable pattern such as $a!\mathsf{Any};a!\mathsf{Any}$. Since the condition of this filter expression is unsatisfiable, it won't match any part of the provenance sequence and hence will not hide any provenance from principals. Note that, strictly speaking, the filter $\mathit{id}_\Phi$ does not give principals full view of the provenance sequence. What it does instead is give principals full view of the provenance relating to the issuer's own actions; access to provenance relating to other principals' actions will depend on what filters were applied by those other principals. This is an intended feature of the filter language as every principal should have full ownership of their own provenance and its disclosure should be subject to their own policies.

**Anonymising filter.** The third filter we define is one that allows principals to post data anonymously. It leaves all provenance relating to other principals intact but completely masks the identify of the poster. It is defined as follows:

$$\mathit{anon}_\Phi = A \circ \mathit{tail}/A$$
$$A = \mathit{author} \circ \mathit{channel}/(\mathit{author} \circ A)$$

This filter is more complicated than the first two and shows a bit more of the features of our filter language. To see that it correctly allows a principal to post data anonymously, consider that a provenance sequence will always contain an output event and an input event after data is posted by the principal, as shown below:

$$a!(a?\ldots ; b!\ldots) ; a?(a?\ldots) ; \ldots$$

As can be seen from the example sequence above, the identity of the poster does not only appear as the author of the receive and send events. It may also appear as the author in the provenance of the channels used to receive and send, and so on in the provenance of the nested channels. To correctly mask the identity of $a$ in the example above, the filter needs to mask all shown occurrences of $a$ as well as those deeper in the provenance of the two channels. The filter $anon_\Phi$ achieves that since it masks the author of the output event, then selects the provenance of the channel used in the output, and masks the author there too, then does this recursively. The same is done for the input event too; it is selected using the *tail* selector, then its author is masked and the same recursive calls are made to ensure that any occurrence of $a$ as author in the provenance of the channel is masked.

**Hide my tracks.** The anonymising filter we defined above hides the name of the principal from the last two events, allowing the principal to post data anonymously. However, if the principal's name appears in earlier parts of the provenance sequence, it will still be visible to other principals (unless masked by other filters of course). We can define a stronger filter which masks the name of the principal anywhere it appears in the provenance sequence. This is given below.

$$\begin{aligned} hidemytracks_\Phi(a) &= [\mathsf{AnyEventBy}(a) ; \mathsf{Any}]A \circ tail/hidemytracks_\Phi(a) \\ A &= author \circ channel/(author \circ A) \\ \mathsf{AnyEventBy}(a) &= a!\mathsf{Any} \vee a?\mathsf{Any} \end{aligned}$$

The definition of the filter $hidemytracks_\Phi(a)$, where $a$ is the name of the principal to hide, relies on the pattern $\mathsf{AnyEventBy}(a)$. This latter matches any event whose author is $a$ and is used as part of the condition in the filter to ensure that only the name of principal $a$ gets hidden. After hiding $a$ from the head of the sequence using the filter expression $A$, a recursive call is made to do the same for the rest of the sequence.

**Hide my sources.** Another useful filter expression is $hidemysources_\Phi$, which as the name suggests, allows a principal to hide the sources of its data. This includes the provenance of the value before it reached the principal. It also includes the provenance of the channels used to receive and send the value before they reached the principal (and the same for any nested channel provenance). The definition of $\mathit{hidemysources}_{\Phi}$ is given below.

$$\mathit{hidemysources}_{\Phi} = A \circ \mathit{tail}/A \circ \mathit{tail}/\mathit{tail}/\mathit{all}$$
$$A = \mathit{channel}/(\mathit{tail} \circ A)$$

The definition above achieves this as follows:

- The call to the recursively defined filter name $A$ allows us to mask the provenance of the channel used to send the value, as well as that of any nested channel. We only mask provenance that was added to the channels before they reached the current principal.
- The call to $\mathit{tail}/A$ does the same as the above but for the channel used to receive the value.
- The filter expression $\mathit{tail}/\mathit{tail}/\mathit{all}$ hides any provenance that was added the value before it reached the current principal.

#### 7.3.3.2 Example systems

**Anonymous proxy.** Consider a system $S$ composed of two principals $a$ and $b$ defined as follows:

$$S = a[m\langle p\rangle] \mid b[m(\pi_b \text{ as } x).P]$$
$$\pi_b = \{a, b, c, \ldots\}!\mathsf{Any}\ ;\ \mathsf{Any}$$

As can be seen from its provenance policy $\pi_b$, principal $b$ only accepts data from a certain set of principals ($\{a, b, c, \ldots\}$). This set happens to include $a$ and so $a$ can send data to $b$ directly if it keeps its identity visible to $b$. This means the following transition is possible.

$$S \Rightarrow_f b[P\{^{p:b?\epsilon;a!\epsilon}/_x\}]$$

However, $a$ may not wish to reveal its identity to $b$. In this case, another way for it to send data to $b$ but still hide its identity is to use another principal trusted by $b$ as a proxy. The following system achieves that.

$$S = a[n\langle p \,\mathsf{at}\, \phi_a\rangle] \mid c[n(x).m\langle x\rangle] \mid b[m(\pi_b \text{ as } x).P]$$
$$\phi_a = \{a, c\} \mapsto_{\mathsf{casc}} \mathit{id}_{\Phi}\ ;\ (\sim - \{a, c\}) \mapsto_{\mathsf{casc}} \mathit{off}_{\Phi}$$

This new system can still make a similar transition to the above, in which $b$ ends up with the value $p$ but $a$'s identity remains hidden from it. The following transition illustrates this.

$$\begin{aligned} S \quad &\Rightarrow_f c[m\,\langle p : c?\epsilon\,;\,\phi_a \triangleright a!\epsilon\rangle] \mid b[m\,(\pi_b \text{ as } x).P] \\ &\Rightarrow_f b[P\{{}^{p:b?\epsilon;c!\epsilon;c?\epsilon;\phi_a \triangleright a!\epsilon}/_{x}\}] \end{aligned}$$

Here, $a$ uses the filter $\phi_a$ to hide the entire provenance of the value $p$ (including $a$'s identity) from every principal except $a$ itself and $c$. This way $c$ can verify the source of the value if it needs to. Principal $c$ then simply forwards it to $b$, who consumes it happily since it trusts $c$. We could have achieved a similar effect in a number of other ways. For example, by using the weaker filter $anon_\Phi$ at $a$ to hide the identity of $a$ only and leave any other provenance visible, or by letting $c$ decide how best to hide the identity of $a$ from $b$ before forwarding the value.

**Conference.** The second example we give models a simplified version of the paper reviewing process at a conference. In this simplified model, the program committee is represented by a single principal whose job is to announce the conference call for papers, collect papers submitted by authors and forward them to selected referees for review. Once the papers are reviewed, the program committee collates this information and decides which papers to accept to the conference. The reviews and committee decisions are then communicated to the authors of papers. We assume a double blind reviewing process where the identities of a paper's authors are kept hidden from the reviewers of the paper and the identities of the reviewers are kept hidden from the authors of the paper. This has to be ensured by the program committee.

**System overview.** At a high level, the conference can be modelled as a system **Conf** composed of three subsystems: **PC** (representing the program committee of the conference), **Auth** (representing one or more authors of papers submitted to the conference), and **Ref** (representing one or more referees responsible for reviewing submissions).

$$\mathbf{Conf} = \mathbf{PC} \mid \mathbf{Auth} \mid \mathbf{Ref}$$

To keep things simple, we model the program committee as a single principal and we assume that we only have 3 authors (each writing and submitting a single paper) and 2 referees (each reviewing all 3 papers).

**Authors.** We start by modeling the behaviour of authors as follows.

$$\textbf{Auth} = \Pi_{i\in\{1,2,3\}} auth_i[ann\,(x).x\,\langle paper_i, r_i\rangle \mid r_i\,(y_p, y_{rs}).P_i]$$

Authors receive the announcement that the conference is open for paper submissions on channel *ann*. With the announcement, they receive the channel name on which the submissions should be made. Each author $auth_i$ then submits their paper $paper_i$ together with a return channel $r_i$ on which the program committee can send back the paper and its reviews to the author.

**Referees.** The behaviour of the two referees is described below.

$$\textbf{Ref} = \Pi_{i\in\{1,2\}} ref_i[\Pi_{j\in\{1,2,3\}} req_i\,(x_j).rev_i\,\langle x_j, review_i(x_j)\rangle]$$

Each referee $ref_i$ receives the three papers to review on channel $req_i$. The referee then sends back the papers together with their reviews to the program committee on channel $rev_i$. Associated with each referee $ref_i$ is a function $review_i$ which gives their review of paper $p$ as $review_i(p)$.

**Program committee.** Now we describe the behaviour of the program committee. As we have seen, the authors and referees do not employ any filters to hide the provenance of data they send to the program committee. Instead, they rely on this latter to mask their identities as required to ensure that the reviewing process retains its anonymity properties.

$$\begin{aligned}\textbf{PC} = pc[\Pi_{i\in\{1,2,3\}}((\nu n)ann\,\langle n\rangle \mid n\,(x_i, y_i).\\ \Pi_{j\in\{1,2\}}(req_j\,\langle x_i \text{ at } \phi'\rangle \mid rev_j\,(x_p, x_r).y_i\,\langle x_p \text{ at } \phi'', x_r \text{ at } \phi''\rangle))]\end{aligned}$$

The program committee announces the conference is open for paper submissions on channel *ann*. Each paper author that listens for this announcement will receive a fresh channel $n$ on which to send back their paper. The program committee accepts three paper submissions, and forwards each paper it receives to the two referees on channels $req_1$ and $req_2$. When sending the papers to the referees, the program committee applies the filter $\phi'$ to their provenance to mask the identity of the authors from the referees. It then listens on channels $rev_1$ and $rev_2$ for the response from the referees. The responses from the referees consist of the papers together with their reviews. Each pair of these gets forwarded to the respective author of the paper, making sure of course that any provenance information that may reveal the identities of the reviewers has been masked appropriately. This is done using the filter $\phi''$. The definitions of the two filters $\phi'$ and $\phi''$ are given below.

**Filters.** We know what each of the filters $\phi'$ and $\phi''$ needs to achieve.

1. Filter $\phi'$ should mask any provenance information that might reveal the identity of a paper's author, in particular to the referees.
2. Filter $\phi''$ should mask any provenance information that might reveal the identity of a paper's reviewer, in particular to its author.

The filter $\phi'$ can be implemented easily by hiding all provenance information that refers to the history of the paper before it reached the program committee. This way the filter $\phi'$ does not depend on the specifics of any particular paper or its history before it reached the program committee. The provenance information should be hidden from everyone except the program committee itself, which will need this information when sending back the reviews to the authors.

$$\phi' = (\sim - pc \mapsto_{\mathsf{over}} \mathit{hidemysources}_{\Phi}) \ ; \ (pc \mapsto_{\mathsf{over}} \mathit{id}_{\Phi})$$

For filter $\phi''$, again to avoid depending on the specifics of the papers or the referees' behaviour, we mask all provenance information that was added after the value was sent from the program committee to the referees. In addition to this, since the reviewing process is complete now, the program committee can reveal the identities of authors. The program committee is able do this since it retained full access to the entire provenance information of the papers and therefore it can use a filter with mode $\mathsf{over}$ to reveal the previously hidden identities of authors.

$$\phi'' = (\sim - pc \mapsto_{\mathsf{over}} \mathit{hidemytracks}_{\Phi}(\mathit{ref}_1) \circ \mathit{hidemytracks}_{\Phi}(\mathit{ref}_2)) \ ; \ (pc \mapsto_{\mathsf{over}} \mathit{id}_{\Phi})$$

**System transitions.** Our conference system **Conf** proceeds as follows. First, the authors receive the announcement that the conference is open for paper submissions.

$$\mathbf{Conf} \Rightarrow_f \mathbf{Conf'} = (\nu n_1, n_2, n_3)(\mathbf{PC'} \mid \mathbf{Auth'} \mid \mathbf{Ref})$$
$$\mathbf{PC'} = pc[\Pi_{i\in\{1,2,3\}}(n_i\,(x_i, y_i).$$
$$\Pi_{j\in\{1,2\}}(req_j\,\langle x_i \,\mathsf{at}\, \phi'\rangle \mid rev_j\,(x_p, x_r).y_i\,\langle x_p \,\mathsf{at}\, \phi'', x_r \,\mathsf{at}\, \phi''\rangle))]$$
$$\mathbf{Auth'} = \Pi_{i\in\{1,2,3\}} auth_i[n_i\,\langle paper_i, r_i\rangle \mid r_i\,(y).P_i]$$

Then, the authors submit their papers to the program committee for consideration.

$$\mathbf{Conf}' \Rightarrow_f \mathbf{Conf}'' = \mathbf{PC}'' \mid \mathbf{Auth}'' \mid \mathbf{Ref}$$
$$\mathbf{PC}'' = pc[\Pi_{i\in\{1,2,3\}}(\Pi_{j\in\{1,2\}}(req_j\,\langle paper_i \;\mathsf{at}\; \phi'\rangle \mid rev_j\,(x_p, x_r).r_i\,\langle x_p \;\mathsf{at}\; \phi'', x_r \;\mathsf{at}\; \phi''\rangle))]$$
$$\mathbf{Auth}'' = \Pi_{i\in\{1,2,3\}} auth_i[r_i\,(y).P_i]$$

Then, the program committee forwards the papers to the referees for review.

$$\mathbf{Conf}'' \Rightarrow_f \mathbf{Conf}''' = \mathbf{PC}''' \mid \mathbf{Auth}'' \mid \mathbf{Ref}'$$
$$\mathbf{PC}''' = pc[\Pi_{i\in\{1,2,3\}}(\Pi_{j\in\{1,2\}}(rev_j\,(x_p, x_r).r_i\,\langle x_p \;\mathsf{at}\; \phi'', x_r \;\mathsf{at}\; \phi''\rangle))]$$
$$\mathbf{Ref}' = \Pi_{i\in\{1,2\}} ref_i[\Pi_{j\in\{1,2,3\}} rev_i\,\langle paper_j, review_i(paper_j)\rangle]$$

After that, the referees review the papers and send them back to the program committee.

$$\mathbf{Conf}''' \Rightarrow_f \mathbf{Conf}'''' = \mathbf{PC}'''' \mid \mathbf{Auth}'' \mid \mathbf{Ref}''$$
$$\mathbf{PC}'''' = pc[\Pi_{i\in\{1,2,3\}}(\Pi_{j\in\{1,2\}}(r_i\,\langle paper_i \;\mathsf{at}\; \phi'', review_j(paper_i \;\mathsf{at}\; \phi'')\rangle))]$$
$$\mathbf{Ref}'' = 0$$

Finally, the program committee sends the papers and their reviews to the authors, announcing which papers have been accepted.

$$\mathbf{Conf}'''' \Rightarrow_f \mathbf{PC}'''' \mid \mathbf{Auth}''' \mid \mathbf{Ref}''$$
$$\mathbf{Auth}''' = \Pi_{i\in\{1,2,3\}} auth_i[P_i\{{}^{paper_i}/_{x}\}\{{}^{(review_1(paper_i), review_2(paper_i))}/_{y}\}]$$
$$\mathbf{PC}''''' = 0$$
$$\mathbf{Ref}'' = 0$$

## 7.4 Concluding remarks

This chapter introduced two concepts, weakenings and filters. These two allowed us to split the problem of controlling the disclosure of provenance into two parts. The first part, addressed by weakenings, considered how the control of provenance disclosure is achieved; while the second, addressed by filters, looked at what programming constructs need to be added to the calculus to implement provenance disclosure policies. The versatility of the pattern language introduced in Chapter 3 and the similarity of its patterns to provenance sequences meant that these patterns could serve as a general notion of weakening. This enabled us to implement a wide range of provenance disclosure policies giving principals full control over what parts of provenance are made available to other principals.

The sample filter language we proposed, while targeting only a subset of weakenings, still allowed us to express a very broad set of examples with varying requirements for provenance disclosure. Not only that, but the combination of over and casc modes offered us a lot of flexibility and allowed us to implement systems where previously hidden provenance may be revealed. The aim of this, as we saw, is to give principals ownership of the provenance of their data and provide them with a way of dynamically changing the visibility of provenance information. All of these points were illustrated by examples in Section 7.3.3.

It is worth noting that the primary driver for this extension of the calculus is security. In Section 7.1, we described how weakenings can also achieve another aim, that of abstracting away from irrelevant details in provenance sequences. We did not consider this when proposing filters however. If removing irrelevant details from provenance sequences is required, then it may still be achievable through filters. One way to do this would be to perform an optimization on systems whereby inaccessible provenance is deleted. Consider for example the filter $\sim \mapsto_{\mathsf{over}} \mathit{off}_{\Phi}$ which when applied to a provenance sequence hides all of it from every principal. Consider the application of this filter to provenance sequence $\kappa$ which results in the filtered provenance sequence $(\sim \mapsto_{\mathsf{over}} \mathit{off}_{\Phi}) \triangleright \kappa$. This provenance sequence would be seen as Any by every principal and there is no way for any principal to change that. Hence, there is no reason to keep the original provenance sequence, and a run time optimization may simply replace it with Any without affecting the behaviour of the system. We do not pursue this further in this work as it is orthogonal to our aims.

As explained already, the tracking of provenance in the original calculus is performed at run time by the operational semantics. Recall that the aim of the operational semantics is to provide a formal specification of the behaviour of the underlying infrastructure. This infrastructure may take the form of a run time environment, an operating system or even a combination of software and hardware components. As an implementation of the operational semantics, the underlying infrastructure is therefore responsible for handling communication between principals and tracking the provenance of data. The infrastructure guarantees the integrity of provenance information and is assumed to be trustworthy. This same assumption is also present in the extension of the calculus described in this chapter. Furthermore, the infrastructure is also responsible for implementing filters and enforcing their security requirements. There are likely to be several ways of doing this depending on how provenance tracking, storage and access are implemented. One way to implement these latter is for the infrastructure to track provenance, store it in a secure provenance repository and only pass references it to principals. When a principal makes a provenance query, which in our calculus takes the form of a pattern on an input action, the infrastructure checks the provenance sequence referenced against the pattern and returns a boolean true or false depending on whether the provenance sequence satisfies the pattern or not. The principal has no access to the actual provenance information. This makes implementing filters easy. The infrastructure stores the filtered provenance sequence (that is the provenance sequence with all the filters imposed on it) in the provenance repository. When a principal makes a provenance query, the infrastructure calculates the principal's view of the referenced provenance sequence, checks this view against the query and returns the results to the principal. Again, queries are simply patterns and therefore their results are simply a boolean specifying whether the pattern is satisfied by the principal's view or not.

# Chapter 8

# Domain Bisimulation

In the extension of the provenance calculus presented in the previous chapter, principals are able, through the application of filters, to associate different views of the same provenance sequence with different principals. The filters specified by a particular principal can be thought of as the *implementation* of an underlying *provenance disclosure policy*, which itself is left implicit. This is similar to patterns; patterns as we saw in chapters 3 and 4 provide the implementation of the principal's *provenance consumption policy*. These two are complementary, patterns allow principals to use available provenance information to decide which data to consume while filters allow them to decide who may access the provenance of data they produce. The behavioural equivalences proposed in Chapter 4 allowed us to study the role of patterns and provenance tracking in the calculus at a more abstract level. Similarly, with the aim of providing a more declarative way to talk about provenance disclosure policies and the role of filters, we define in this chapter a notion of behavioural equivalence for the filtered calculus. We make this equivalence parametric on the set of principals comprising the observer. By varying the set of principals, and hence the privileges, available to the observer, we should be able to specify the requirements of provenance disclosure policies and verify the correctness of their implementations as sets of filters.

We first start by reviewing the version of the calculus defined in the previous chapter and providing an alternative formulation of its semantics in the form of a labelled transition system. This is covered in Section 8.1. After that, in Section 8.2, we define domain congruences, a family of behavioural equivalences that is sensitive to the domain of the observer. This means that the discriminating power of the equivalence depends on the set of principals available to the observer. The reason for making the equivalence parametric on the domain of the observer is to be able to reflect, in the equivalence, the different views assigned to different principals by filters. That way, we can use the equivalences to specify the desired aims of a principal's provenance disclosure policy and check whether the filters used by the principal achieve those aims. Finally, in Section 8.3, we provide a characterisation of domain equivalence based on the labelled transition system semantics of the calculus. Not only does this provide us with an efficient way to prove domain equivalence, it also sheds more light on its properties and the role of filters in the calculus. Section 8.4 provides some concluding remarks.

## 8.1 The provenance calculus with filters

Chapter 7 extended the provenance calculus with filters to give principals control over the provenance information of data they publish. Filters allowed principals to specify how much provenance other principals are able to see and hence gave them a way to control the disclosure of sensitive provenance information. This extension involved changes to the output construct so that principals could specify what filter to apply, as well as changes to the input construct so that principals could only check their policies against their views of provenance and not the entire provenance sequence. We give a brief overview of these changes in this section and then provide an alternative formulation of the semantics of the calculus using a labelled transition system. This alternative semantics will form the basis of the bisimulation defined later in section 8.3. Note that for the purposes of this chapter, we do not need to rely on the specifics of any filter language. Instead, we give our results in terms of the generic description of filter languages given in Definition 7.5. The results should be correct for any concrete filter language, including of course the sample filter language defined in the previous chapter.

### 8.1.1 Syntax and reduction semantics

In terms of syntax, we introduced a new syntactic category for filters, ranged over by the letters $\phi, \psi, \ldots$. We also extended the syntax of provenance sequences with filtered sequences $\phi \triangleright \kappa$ to denote the application of filter $\phi$ to provenance sequence $\kappa$. The output construct was also modified to specify, in addition to the channel name and value, the filter to apply to the provenance of the value being sent. The syntax of filters itself was left unspecified as the calculus was made parametric on the filter language. We do the same here as the exact details of the filter language should not have any bearing on the results presented in this chapter. The rest of the syntax of the calculus remained unchanged from what was given in Chapter 3.

The reduction semantics of the calculus was given in terms of two relations, structural congruence $\equiv_f$ and reduction $\rightarrow_f$. Structural congruence for the extended calculus is the same as that of the original calculus, of course adapted to the new syntax. The reduction relation on the other hand changes to account for the addition of filters. As we saw, only the input and output rules were modified. When sending a value, the filter specified by the principal is applied to the provenance of the value, while when receiving a value, the principal is only allowed to perform pattern matching against their own view of provenance. The previous chapter gives more information about the calculus and filters while Appendix B gives the full definitions of its syntax and reduction semantics.

### 8.1.2 Labelled semantics

In this chapter, we aim to define both a contextual equivalence and a coinductive characterisation of it. In order to be able to define this latter, we need to introduce a labelled version of the semantics of the filtered calculus.

#### 8.1.2.1 Actions

The labelled semantics is given in terms of a family of binary relations on systems of the form $\xrightarrow{\alpha}_f$, where $\alpha$ ranges over actions. We have four kinds of actions as follows:

1. $\overline{m}\langle p{:}\kappa\rangle$: which denotes the output of the *free* annotated value $p : \kappa$ on channel $m$.
2. $\overline{m}(n{:}\kappa)$: which denotes the output of the *bound* annotated name $n : \kappa$ on channel $m$.
3. $m\langle p{:}\kappa\rangle$: which denotes the input of the *free* annotated value $p : \kappa$ on channel $m$.
4. $\tau$: which denotes the unobservable (internal) action.

As indicated above, the name $n$ in action $\overline{m}(n : \kappa)$ is bound; all other occurrences of names in actions are free. We use $bn(\alpha)$ and $fn(\alpha)$ for the sets of bound names and free names in action $\alpha$ respectively. We denote the union of these two by $n(\alpha)$.

#### 8.1.2.2 Rules of the labelled transition system

While the reduction semantics gives an internal view of the behaviour of systems, expressing how communication happens within a system; the labelled semantics gives an external view of it, expressing how a system could potentially communicate if put in a suitable environment. Internal actions, labelled with $\tau$, indicate the communication of some value within the system itself, and hence should correspond to system transitions in the reduction semantics. The external environment would not be expected to affect or perceive internal actions beyond merely observing their occurrence. External actions on the other hand, labelled with input or output labels as described already, indicate the ability of the system to engage in communication if the environment were to perform a complementary action. This requirement on the side of the external environment means that the environment is able to observe exactly what took place in the case of external actions.

The family of transition relations $\xrightarrow{\alpha}_f$, where $\alpha$ ranges over actions, is defined by the rules of Figure 8.1.

Figure 8.1. *Filtered semantics: labelled transition system*

FAct Snd

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\ \mathsf{at}\ \phi\rangle] \xrightarrow{\tau}_f m\langle\!\langle p : \phi \triangleright a!\kappa_m\ ;\ \kappa_p\rangle\!\rangle}$$

FAct Msg

$$\frac{}{m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \xrightarrow{\overline{m}\langle p:\kappa_p\rangle}_f 0}$$

FAct Rcv

$$\frac{j \in I \quad \mathit{view}(a, \kappa_p) \leq \pi_j}{a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i\ \mathsf{as}\ x).P_i] \xrightarrow{m\langle p:\kappa_p\rangle}_f a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

FAct If-t

$$\frac{}{a[\mathsf{if}\ m{:}\kappa = m{:}\kappa'\ \mathsf{then}\ P\ \mathsf{else}\ Q\ ] \xrightarrow{\tau}_f a[P]}$$

FAct If-f

$$\frac{m \neq n}{a[\mathsf{if}\ m{:}\kappa = n{:}\kappa'\ \mathsf{then}\ P\ \mathsf{else}\ Q\ ] \xrightarrow{\tau}_f a[Q]}$$

FAct Exp

$$\frac{a[P] \xrightarrow{\alpha}_f S \quad n \notin n(\alpha)}{a[(\nu n)P] \xrightarrow{\alpha}_f (\nu n)S}$$

FAct Res

$$\frac{S \xrightarrow{\alpha}_f S' \quad n \notin n(\alpha)}{(\nu n)S \xrightarrow{\alpha}_f (\nu n)S'}$$

FAct Open

$$\frac{S \xrightarrow{\overline{m}\langle n\rangle}_f S'}{(\nu n)S \xrightarrow{\overline{m}(n)}_f S'}$$

FAct Par-l

$$\frac{S \xrightarrow{\alpha}_f S' \quad bn(\alpha) \cap fn(T) = \emptyset}{S \mid T \xrightarrow{\alpha}_f S' \mid T}$$

FAct Comm-l

$$\frac{S \xrightarrow{\overline{m}\langle p:\kappa\rangle}_f S' \quad T \xrightarrow{m\langle p:\kappa\rangle}_f T'}{S \mid T \xrightarrow{\tau}_f S' \mid T'}$$

FAct Close-l

$$\frac{S \xrightarrow{\overline{m}(n:\kappa)}_f S' \quad T \xrightarrow{m(n:\kappa)}_f T'}{S \mid T \xrightarrow{\tau}_f (\nu n)(S' \mid T')}$$

FAct Spl-l

$$\frac{a[P] \xrightarrow{\alpha}_f S}{a[P \mid Q] \xrightarrow{\alpha}_f S \mid a[Q]}$$

FAct Rep

$$\frac{a[P] \xrightarrow{\alpha}_f S}{a[* P] \xrightarrow{\alpha}_f S \mid a[* P]}$$

The rules in Figure 8.1 describe what actions a particular system is able to perform. The rule FAct Snd expresses that a principal $a[m{:}\kappa_m \langle p{:}\kappa_p \,\mathsf{at}\, \phi\rangle]$ may perform an internal action $\tau$ which would result in the message $m\langle\!\langle p : \phi \triangleright a!\kappa_m \,;\, \kappa_p\rangle\!\rangle$. This is actually the same as FRed Snd. As previously stated, this is to be expected as internal actions in the labelled semantics are meant to mirror the transitions of the reduction relation. The message produced by the output system may be consumed by a system listening on the same channel like we saw in the reduction semantics. In the labelled semantics, this is expressed by a number of rules. Firstly, the production of a value to be consumed is expressed by the rule FAct Msg, which states that a message $m\langle\!\langle p{:}\kappa_p\rangle\!\rangle$ may perform the action $\overline{m}\langle p{:}\kappa_p\rangle$, indicating the output of free value $p{:}\kappa_p$ on channel $m$. Secondly, a system listening on channel $m$ may perform the input action $m\langle p{:}\kappa_p\rangle$, inputting a value with suitable provenance and resulting in the system $a[P_j\{{}^{p:a?\kappa_m;\kappa_p}/_{x}\}]$. The two rules FAct Msg and FAct Rcv give the two halves of communication expressed by the rule FRed Rcv in the reduction semantics. So finally, to complete the communication transition, these two halves may be joined together by the rule FAct Comm-l, indicating the communication of a value from one part of the system to another. This, from the point of view of the external environment, is perceived as an internal action $\tau$ with no further details on what value was actually communicated or what channel it was communicated on. Note that for some rules, we have a left and a right variant, which differ only in the position of the two systems involved. These rules have the suffix "-l" to distinguish them from others. We only included the left versions in the above figure; the right versions can be obtained by swapping the positions of the two systems in the left version. The remaining rules of the labelled transition system are for the most part adaptations of those of the asynchronous $\pi$-calculus to our setting. The reader is referred to the literature for more information on labelled transition systems in general as well as to their use in the $\pi$-calculus in particular.

#### 8.1.2.3 Correspondence between the two semantics

The labelled transition system of Figure 8.1 is meant to provide an *alternative* formulation of the semantics of the filtered calculus, and as such, we expect it to coincide with the reduction semantics which we reviewed briefly in the previous section. This is shown formally in Theorem 8.5. To prove this latter as well as other results below, we need to first prove a number of properties concerning the labelled transition system and its relation to the reduction semantics. We do this below. Note that we omit the proofs of these lemmas as they are simply adaptations of equivalent results from the literature to our setting.

**Lemma 8.1.** *$S \equiv_f T$ and $T \xrightarrow{\alpha}_f T'$ implies that $S \xrightarrow{\alpha}_f S'$ for some $S'$ such that $T' \equiv_f S'$.*

**Lemma 8.2.** *$S \xrightarrow{\tau}_f S'$ implies that $S \longrightarrow_f S'$.*

**Lemma 8.3.** *The labelled semantics satisfies the following properties:*

- *$S \xrightarrow{\overline{m}\langle v\rangle}_f S'$ implies that $S \equiv_f m\langle\!\langle v\rangle\!\rangle \mid S'$.*
- *$S \xrightarrow{\overline{m}(n)}_f S'$ implies that $S \equiv_f (\nu n)(m\langle\!\langle n\rangle\!\rangle \mid S')$.*

**Lemma 8.4.** *$S \longrightarrow_f S'$ implies that $S \xrightarrow{\tau}_f S''$ for some $S''$ such that $S' \equiv_f S''$.*

**Theorem 8.5.** *$S \longrightarrow_f S'$ if and only if $S \xrightarrow{\tau}_f S''$ and $S'' \equiv_f S'$.*

*Proof.* Follows from Lemma 8.2 and Lemma 8.4. □

## 8.2 Contextual equivalence

Filters give principals ownership of the provenance information and allow them to specify how much of it they wish to be made available to other principals. Even more, they let a principal classify other principals into different groups and give each group its own view of provenance. The classification of principals into groups would be driven by the properties of the underlying domain being modelled, such as the trustworthiness attributed to different principals, their assigned clearance levels, or their registered interests. What exactly drives this classification in a particular system, and indeed the classification itself and what it entails in terms of provenance disclosure, are left implicit however. What we are able to see instead is how this is implemented by filters. Because filters are "embedded in the code", it is hard to tell from them what the intended provenance disclosure policy of the principal is. Moreover, the absence of a high level specification of a principal's provenance disclosure policy, and of a method to relate it to its low level implementation by a set of filters, means that there is no way for us to tell if the filters correctly implement the desired policy of the principal. Our aim in this section is to address this by proposing a notion of behavioural equivalence that could serve such a purpose.

### 8.2.1 Provenance disclosure policies

Fundamentally, provenance disclosure policies, and by extension filters as these latter are no more than the implementations of disclosure policies, assign each principal or group of principals a view of the provenance information. Hence, in order for us to be able to reason about the role of filters, we make our notion of behavioural equivalence parametric on the set of principals available to the observer. That way, we can vary the set of principals making up the observer and look at how that affects the equivalence. What that gives us, in effect, is a more declarative higher level way of looking at the provenance view assigned to each principal. To illustrate this, consider for example a principal $a$ running process $P$ and whose provenance disclosure policy dictates that they should remain anonymous in all their dealings with principals $b$ and $c$. To ensure this, we could analyse the process $P$ running at $a$ and make sure that in all its interactions with the outside world it never does anything that could divulge its identity to $b$ or $c$. This means verifying all patterns and filters used in $P$ and ensuring that they effectively hide $a$'s identity from $b$ and $c$. Even for such a simple policy, the analysis would not be straightforward. Using our notion of equivalence on the other hand, anonymity with respect to principals $b$ and $c$ could be stated simply as:

$$a[P] \;\cong_{b,c}\; a[P]\sigma \quad \text{for all renamings } \sigma$$

What the above says is that the system $a[P]$ is indistinguishable to principals $b$ and $c$ from the system $a[P]\sigma$. This latter represents the result of renaming $a$ in $a[P]$ to some other principal name. Since $b$ and $c$ (no matter what processes they are running) can not distinguish between $a[P]$ and $a'[P]$ (or $a''[P]$ or any system that we would get by applying a renaming $\sigma$ to $a[P]$), then we know that the filters in $P$ correctly hide the identity of $a$ from $b$ and $c$.

The two principals $b$ and $c$ in the equivalence $\cong_{b,c}$ can be thought of as the two principals comprising the observer. That is, the observer, working to distinguish between the two systems $a[P]$ and $a[P]\sigma$, may run any process under the authority of principals $b$ and $c$. That allows the observer to view any provenance made available to these two principals. We refer to sets of principal names such as $\{b, c\}$ as *domains*, and refer to notions of behavioural equivalence such as $\cong_{b,c}$ that are parametric on such sets of principal names as *domain congruences*. Note that the notion of observer used in domain congruences is the traditional one used in most definitions of bisimulation. That is, an observer is a term in the calculus and therefore has no additional information than that obtained from interacting with the systems under observation according to the rules of the calculus.

Note also that since we are making the equivalence parametric on the domain of the observer, what we actually have is a family of equivalence relations, one for each domain.

### 8.2.2 Domain congruences

Domain congruences are an adaptation of the standard framework of *barbed bisimulation and congruence* [50, 64] to our setting. Like other barbed congruences, they equate two systems if they exhibit the same observable behaviour, evolve to equivalent systems, and cannot be distinguished when put in the same context. The difference, however, is in the contexts we choose; each domain congruence is identified by a set of principals and required to be closed under contexts of that set.

#### 8.2.2.1 Domains

The domain of a system is simply the set of principals it contains. This is important since filters associate views of provenance with principals, and therefore knowing the domain of a system allows us to determine how much provenance is available to it. Given a system $S$, its domain is denoted by $dom(S)$ and is defined inductively on the structure of system $S$ as given in Definition 8.6. We extend the notion of domains to contexts in 8.7. To make it easier to group systems and contexts based on their domains, we define the notion of $D$-systems and $D$-contexts. A system $S$ is said to be a $D$-system if its domain is a subset of the set of principal names $D$. Again, $D$-contexts are defined similarly. The formal definition of $D$-systems and $D$-contexts is given in Definition 8.8.

**Definition 8.6** (Domain of a system)**.** The domain of a system is defined as follows:

$$dom(0) = \emptyset \qquad dom(m\langle\!\langle v \rangle\!\rangle) = \emptyset \qquad dom(a[P]) = \begin{cases} \{a\} & \text{if } a[P] \not\equiv_f 0 \\ \emptyset & \text{if } a[P] \equiv_f 0 \end{cases}$$

$$dom((\nu n)S) = dom(S) \qquad dom(S \mid T) = dom(S) \cup dom(T)$$

**Definition 8.7** (Domain of a context)**.** The domain of a context is defined as follows:

$$dom([.]) = \emptyset \qquad dom((\nu n)C) = dom(C)$$

$$dom(S \mid C) = dom(S) \cup dom(C) \qquad dom(C \mid S) = dom(C) \cup dom(S)$$

**Definition 8.8** ($D$-systems and $D$-contexts)**.** Let $D$ be a domain. A $D$-system is a system $S$ whose domain is a subset of $D$, i.e., $dom(S) \subseteq D$. Similarly, a $D$-context is a context $C$ whose domain is a subset of $D$, i.e., $dom(C) \subseteq D$.

Note that $dom(-)$ is preserved under structural congruence. That is, for any two systems $S$ and $T$, the fact that $S \equiv_f T$ holds implies that $dom(S) = dom(T)$ also holds. This is the case because the clause for $dom(a[P])$ guarantees that any inactive principals are not included in the domain of the system. To illustrate this, consider the following example:

$$a[m\langle v\rangle] \equiv_f a[m\langle v\rangle] \mid b[0] \equiv_f a[m\langle v\rangle] \mid b[0] \mid c[0]$$

Although all three systems above contain different principals, their domains are in fact the same because they only differ in the inactive principals $b[0]$ and $c[0]$. These two both have the empty set as their domain, and therefore, the domains of all three systems are the set $\{a\}$. Proposition 8.9 shows that the notion of $D$-systems (and by extension $D$-contexts) is preserved under structural congruence.

**Proposition 8.9.** *Let $S$ and $T$ be systems. If $S \equiv_f T$ holds, then for all domains $D$, it holds that $S$ is a $D$-system if and only if $T$ is a $D$-system.*

*Proof.* Let us assume that $S \equiv_f T$. The proof proceeds by rule induction on the derivation of $S \equiv_f T$. In each case, it should be obvious that $dom(S) = dom(T)$. In fact, the only case where the set of principals appearing in the systems changes is $a[0] \equiv_f 0$. As we have already explained, both of these have the domain $\emptyset$. In all other cases, the set of principals stays the same. Since the domains of $S$ and $T$ are the same, then they are subsets of exactly the same sets. Therefore, system $S$ is a $D$-system if and only if system $T$ is a $D$-system. □

#### 8.2.2.2 The family of domain congruences

The observation predicate, denoted by $S \downarrow_f m$ and defined formally in Definition 8.10, states that channel name $m$ is observable at system $S$. What this means is that a system that is interacting with system $S$, particularly an observer, can expect system $S$ to send some value on channel $m$. Therefore, all that is needed for the observer to interact with $S$ is to listen on channel $m$. Other than listening for outputs from the systems under observation, the observer has no other way of telling systems apart. In particular, it has no information about the internal structure of the systems, it cannot eavesdrop on internal communication performed on private channels, and it cannot in general tell when a value it sent is received as the calculus only allows asynchronous communication. In addition to this, the provenance infrastructure is assumed to be both trusted and secure; that is, provenance information cannot be forged and filters cannot be circumvented.

**Definition 8.10** (Observation predicate)**.** We say that a channel name $m$ is observable at a system $S$, written $S \downarrow_f m$, if and only if $S \equiv_f (\nu n)(m\langle\!\langle v\rangle\!\rangle \mid S')$ for some name $n$, value $v$ and system $S'$ such that $m \neq n$. We also write $S \Downarrow_f m$ to mean that there exists some system $S'$ such that $S \Longrightarrow_f S'$ and $S' \downarrow_f m$.

Since we are working in an asynchronous setting, only outputs are generally observable by the environment. That is, in most cases, an observer interacting with the system according to the rules of the calculus would not be able to detect when a value it sent has been received by the system. There are special cases under which inputs are observable however and we shall comment on these later. Note that since we use two-step communication in our calculus, what is observable are messages and not to output processes. We only need to consider the channel of the message as the equivalence we obtain is the same as the one that we would get if we had taken more of the message as our basic observable.

We define the family of domain congruences below. Each domain congruence is associated with a domain $D$ and characterises those systems that are perceived as behaviourally equivalent by all observers whose domain is the set $D$. The difference between an observer whose domain is $D$ and an observer whose domain is $D'$ is the amount of provenance they are able to view. Therefore, domain equivalences enable us to reason about the views of provenance assigned to principals by filters. The other aspects of the definition of domain congruences are standard. Two systems are considered to be behaviourally equivalent if they exhibit the same observables, remain equivalent throughout their lifecycles and cannot be distinguished by any context induced by the particular domain congruence.

**Definition 8.11** (Domain congruence)**.** Let $D$ be a domain. $D$-congruence, $\cong_D$, is the largest symmetric binary relation on systems that is:

- barb-preserving: $S \cong_D T$ implies that for all channel names $m$, if $S \downarrow_f m$ then $T \Downarrow_f m$.
- reduction-closed: $S \cong_D T$ implies that if $S \longrightarrow_f S'$ for some $S'$ then $T \Longrightarrow_f T'$ for some $T'$ such that $S' \cong_D T'$.
- $D$-contextual: $S \cong_D T$ implies that $C[S] \cong_D C[T]$ for all initial $D$-contexts $C$.

Restricting contextuality in domain congruences to contexts with a particular domain means that some differences in provenance between systems will not be detectable by the observing environment. An observer whose domain is restricted to principals in a particular set can only discriminate between systems based on the views of provenance made available to principals in that set. Note that in addition to this, we are also restricting contextuality to initial contexts. Recall that a context is initial if all its values have empty provenance. The reason we restrict the equivalence to initial contexts is to be able to account for all provenance that arises in the observer by its interaction with the system under observation. It should be clear that the more principal names that are available to the observer to use, the more provenance the observer is able to see, and hence the finer the resulting domain equivalence is. This is formally stated and proved in Proposition 8.12.

**Proposition 8.12.** *Let $D$ be a domain. For all domains $D'$ such that $D' \subseteq D$, it holds that $\cong_D \subseteq \cong_{D'}$.*

*Proof.* Let $D$ and $D'$ be two domains such that $D' \subseteq D$. To prove that $\cong_D \subseteq \cong_{D'}$ holds, we just need to show that $\cong_D$ satisfies all the properties of $\cong_{D'}$. By definition, $\cong_D$ is symmetric, barb-preserving and reduction-closed. So we just need to show that it is preserved by $D'$-contexts. This is easy to show since $\cong_D$ is preserved by $D$-contexts, and every $D'$ context is by definition a $D$-context. □

The second property we prove about domain congruences is that they are equivalence relations. This will allow us to use equational reasoning to prove the equivalence of systems.

**Proposition 8.13.** *Let $D$ be a domain. The domain congruence $\cong_D$ is an equivalence relation.*

*Proof.* Let $D$ be a domain. We know that $\cong_D$ is symmetric by definition. We need to show that it is also reflexive and transitive; we do that by contradiction. Assume $(S, S) \notin \cong_D$ and consider the relation $\dot{\cong}_D$ defined as $(S, S) \cup \cong_D$. It is easy to show that $\dot{\cong}_D$ satisfies all the defining properties of Definition 8.11 and since it is larger than $\cong_D$, we have reached a contradiction. Hence, it must be the case that $S \cong_D S$. The proof of transitivity is similar. □

#### 8.2.2.3 System properties

With these definitions in place, we can now formally prove properties about systems modelled in the filtered calculus. For instance, we can prove whether the filters employed by principal $a$ in the example given earlier in this section correctly keep its identity anonymous from principals $b$ and $c$ or not. In order to do that, we need to give more concrete details about the example itself. Let us then use the sample filter language of Section 7.3 and define the system $S$ as follows:

$$S \triangleq a[m\langle p \,\mathsf{at}\, \phi\rangle] \quad \text{where } \phi \triangleq \{b,c\} \mapsto_{\mathsf{over}} anon_\Phi \,;\, (\sim - \{b,c\}) \mapsto_{\mathsf{over}} id_\Phi$$

The example above is very simple and makes use of two of the filter idioms we defined in the previous chapter, $anon_\Phi$ and $id_\Phi$. As we saw in the previous chapter, the filter $anon_\Phi$ is an anonymising filter while the filter $id_\Phi$ is an inert filter. Used as given above in the example, these should allow principal $a$ to hide its identify from $b$ and $c$.

**Proposition 8.14.** *Let $S$ be the system defined as follows:*

$$S \triangleq a[m\langle p \,\mathsf{at}\, \phi\rangle] \quad \textit{where } \phi \triangleq \{b,c\} \mapsto_{\mathit{over}} anon_\Phi \,;\, (\sim - \{b,c\}) \mapsto_{\mathit{over}} id_\Phi$$

*It is the case that $S \approxeq_{b,c} S\sigma$ for any renaming $\sigma$.*

*Proof.* In order to show that $S \approxeq_{b,c} S\sigma$, we just need to find a relation $r$ that contains the pair $(S, S\sigma)$ and that exhibits the three defining properties of $\approxeq_{b,c}$. Once we do that, it would follow that $r \subseteq \approxeq_{b,c}$ since $\approxeq_{b,c}$ is by definition the largest relation that satisfies those three properties. Let $r$ be the relation containing pairs of the form $(S, S\sigma)$ and closed under initial $\{b,c\}$-contexts. Relation $r$ is by definition $\{b,c\}$-contextual and therefore we only need to show that it is barb-preserving and reduction-closed. We do this by induction on why the pair $(S,T)$ is in relation $r$. There are two cases, either $(S,T)$ is of the form $(S', S'\sigma)$ for some system $S'$ and renaming $\sigma$, or $(S,T)$ is of the form $(C[S'], C[T'])$ for some initial $\{b,c\}$-context $C$ and systems $S'$ and $T'$ that are in relation $r$. In each case, we need to show that $(S,T)$ have the same observables and make the same reductions. The details should be straightforward. The important thing to note is that the systems $C[S]$ and $C[S\sigma]$ will have exactly the same observables and reductions because the context $C$ will not be able to detect the name of principal $a$ and its renaming $a\sigma$ in the provenance of value $p$. This is a result of the application of filter $\phi$ to the provenance sequence. □

Recall that renamings were one of the methods we used to compare the plain version of the calculus to the annotated one in Chapter 4. There, we saw that since principal identities do not play any role in the semantics of the plain version of the calculus, its behavioural equivalence was invariant under renamings. However, this was not the case in the annotated version as provenance tracking and checking expose the identities of principals to the external environment. The above proposition, on the other hand, shows that under certain circumstances renamings are immaterial in the filtered version of the calculus too. This is interesting as it shows the trade-offs between provenance tracking and filtering. Provenance tracking exposes the identities of principals and their communication history to other principals; while filtering masks that information, positioning the filtered calculus somewhere between the plain version and the annotated version in terms of the availability of provenance.

Renamings are most useful for expressing and proving anonymity properties. Domain congruences can also be used to provide specifications for more general properties however. For example, consider the following system which is supposed to implement a proxy:

$$a[* \, m\,(\textsf{Any as } x).n\,\langle x \,\textsf{at}\, \phi\rangle] \qquad \text{where } \phi = \sim\, \mapsto_{\textsf{over}} \mathit{hidemysources}_{\Phi}$$

This system continuously reads data from channel $m$ and forwards it onto channel $n$. It uses the filter expression $\mathit{hidemysources}_{\Phi}$ to hide the previous provenance of the value it receives on $m$ before it sends it on $n$. A principal can use this system as a proxy when interacting with other principals to hide all provenance of its data. We can express this property of the proxy by the following equivalence:

$$S\,(\kappa_1) \approxeq_{\sim} S\,(\kappa_2)$$

where the system $S$ is defined as follows:

$$S\,(X_\kappa) \triangleq (\nu m)(m\langle\!\langle p : X_\kappa \rangle\!\rangle \mid a[* \, m\,(\textsf{Any as } x).n\,\langle x \,\textsf{at}\, \phi\rangle])$$

What the above equivalence says is that the two systems $S\,(\kappa_1)$ and $S\,(\kappa_2)$ are indistinguishable to every system. System $S\,(X_\kappa)$ represents the proxy attempting to hide the provenance $X_\kappa$. If we can prove that $S\,(\kappa_1)$ and $S\,(\kappa_2)$ are equivalent, then that should constitute irrefutable proof that the proxy $a[* \, m\,(\textsf{Any as } x).n\,\langle x \,\textsf{at}\, \phi\rangle]$ does indeed hide its sources. Many other properties of filters can be expressed in a similar way using domain congruence.

## 8.3 Domain bisimilarity

The definition of domain congruence involves a universal quantification over initial contexts, and this makes it difficult to use for proving the equivalence of systems in general. To obtain a more tractable proof methodology, we define domain bisimilarity. The definition of this latter relies on the labelled transition system of Figure 8.1, which provides an alternative formulation of the semantics of the filtered version of the calculus. We should note that the notion of bisimilarity we propose in this section is proved to be sound but not complete. What that means is that any two systems that are deemed equivalent by a domain bisimilarity, are guaranteed to be equivalent under the corresponding domain congruence. However, the inverse implication does not hold. Provenance and filters interact in subtle ways as the definition of domain bisimilarity will show. We leave it as an open question whether a sound *and complete* coinductive characterisation of domain congruence exists.

### 8.3.1 Observational labelled transition systems

The challenge in defining a bisimulation based on the labelled transition system of Figure 8.1 is that this latter is *not observational.* What we mean by this is that the action labels of the transition system contain information that would not be detectable by an observer interacting with the system according to the rules of the calculus. More specifically, an observer can only see the parts of provenance that have not been filtered out from the views of its principals. This means that we have to consider the principals available to the observer before we can decide whether two output actions are observationally equivalent or not. To illustrate this, consider the following two systems:

$$S \triangleq a[m\,\langle p\,\mathsf{at}\,\phi\rangle] \qquad\qquad T \triangleq b[m\,\langle p\,\mathsf{at}\,\phi\rangle]$$

where $\phi \triangleq \{c\} \mapsto_{\mathsf{over}} anon_{\Phi}\,;\,(\sim - \{c\}) \mapsto_{\mathsf{over}} id_{\Phi}$. The transitions of the two systems can be given as follows:

$$\begin{array}{lll} S & \xrightarrow{\tau}_f & m\langle\!\langle p : \phi \triangleright a!\epsilon \rangle\!\rangle \\ & \xrightarrow{\overline{m}\langle p:\phi\triangleright a!\epsilon\rangle}_f & 0 \end{array} \qquad \begin{array}{lll} T & \xrightarrow{\tau}_f & m\langle\!\langle p : \phi \triangleright b!\epsilon \rangle\!\rangle \\ & \xrightarrow{\overline{m}\langle p:\phi\triangleright b!\epsilon\rangle}_f & 0 \end{array}$$

System $S$ makes a $\tau$ action followed by an output action with the label $\overline{m}\langle p : \phi \triangleright a!\epsilon\rangle$. System $T$ makes a similar $\tau$ action followed by an output action with a different label, $\overline{m}\langle p : \phi \triangleright b!\epsilon\rangle$. Since the two output labels differ in the principal name, a bisimulation that naively compares the labels of the transitions would declare the two systems as inequivalent. However, observers interacting with the two systems do not perceive the output labels as $\overline{m}\langle p : \phi \triangleright a!\epsilon\rangle$ and $\overline{m}\langle p : \phi \triangleright b!\epsilon\rangle$. Instead, observers see the provenance of data as it appears in the views assigned to their principals. So for example, if the observer is only restricted to processes running at principal $c$, then it would actually perceive the two labels as $\overline{m}\langle p : \sim!\epsilon\rangle$ and $\overline{m}\langle p : \sim!\epsilon\rangle$. Therefore, observers whose domain is $\{c\}$ would have but to declare systems $S$ and $T$ as equivalent. On the other hand, observers with access to other principals, say $\{d\}$ for example, would be able to observe the entire provenance, perceiving the two output labels as $\overline{m}\langle p : a!\epsilon\rangle$ and $\overline{m}\langle p : b!\epsilon\rangle$. This implies that such observers would be able to tell the two systems $S$ and $T$ apart.

Comparing the views of provenance assigned to the observer is not enough to give us a bisimulation that is consistent with domain congruences however. The reason is that even when the observer is not able to directly access the provenance information, it could feed the values it receives back to the two systems under testing and observe how they react. If the reactions of the two systems are different, the observer can infer the reason to be most probably hidden provenance. Either case, the observer would be able to discern a difference between the two systems and therefore declare them as inequivalent. Hence, a bisimulation based on the labelled transition system need not only consider the views of provenance available to the observer, but also those available to the two systems themselves. The latter is important since it is possible that, by accessing provenance that is hidden from the observer, the two systems may unintentionally leak it.

### 8.3.2 Equivalence of provenance

Two provenance sequences, $\kappa_1$ and $\kappa_2$ may be indistinguishable to a principal $a$ if they associate the same view with principal $a$. That is, if $\mathit{view}(a, \kappa_1) = \mathit{view}(a, \kappa_2)$. We generalise this to sets of principals by defining $D$-equivalence.

**Definition 8.15.** Let $D$ be a domain. Two provenance sequences $\kappa_1$ and $\kappa_2$ are said to be $D$-equivalent, written $\kappa_1 =_D \kappa_2$, if for all $a \in D$, it holds that $\mathit{view}(a, \kappa_1) = \mathit{view}(a, \kappa_2)$.

When considering whether two systems are equivalent or not, we have to take into account the provenance annotations of values they publish. If these provenance annotations are meant to be indistinguishable to the observer, then the two systems under observation need to make sure they do not inadvertently leak any hidden provenance if those values are fed back to them. To allow us to keep track of the different provenance sequences published by the two systems in the bisimulation, we use *equivalence environments*. Equivalence environments are used to keep track of pairs of provenance sequences that the observer is meant to perceive as being equivalent. As the two systems under observation intend to convince the observer that any pair of provenance sequences in the equivalence environment are the same, they need to "pretend" that these two are indeed equivalent in all their interactions with the observer. We define $D$-environments to be equivalence environments where every pair of provenance sequences are $D$-equivalent.

**Definition 8.16.** An equivalence environment is a set of pairs of provenance sequences. We use the letter $\Delta$ and its variants to denote equivalence environments. We define a $D$-environment to be one where every pair of provenance sequences $(\kappa_1, \kappa_2)$ are $D$-equivalent, i.e., where it is the case that $\kappa_1 =_D \kappa_2$.

For a given domain $D$, every pair of provenance sequences in a $D$-environment are considered equivalent. In addition to this, a $D$-environment also yields other equivalent pairs by inference. More specifically, any two provenance sequences of which the pairs in the $D$-environment are part should also be considered equivalent. We capture this in the definition of equivalence under an environment given below.

**Definition 8.17.** Let $D$ be a domain. Two provenance sequences $\kappa_1$ and $\kappa_2$ are said to be $D$-equivalent *under environment* $\Delta$, written $\Delta \vdash \kappa_1 =_D \kappa_2$, if such a judgement could be derived from the axioms and rules of Figure 8.2.

Figure 8.2. *Definition of* $\Delta \vdash \kappa_1 =_D \kappa_2$

Prov Eps

$$\frac{}{\Delta \vdash_a \epsilon =_D \epsilon}$$

Prov out

$$\frac{a \in D \quad \Delta \vdash_a \kappa_1 =_D \kappa_2 \quad \Delta \vdash_a \kappa'_1 =_D \kappa'_2}{\Delta \vdash a!\kappa_1 \,;\, \kappa'_1 =_D a!\kappa_2 \,;\, \kappa'_2}$$

Prov In$_1$

$$\frac{\Delta \vdash_a \kappa_1 =_D \kappa_2 \quad \Delta \vdash \kappa'_1 =_D \kappa'_2}{\Delta \vdash_a a?\kappa_1 \,;\, \kappa'_1 =_D a?\kappa_2 \,;\, \kappa'_2}$$

Prov In$_2$

$$\frac{\Delta \vdash_a \kappa_1 =_D \kappa_2 \quad (\kappa'_1, \kappa'_2) \in \Delta}{\Delta \vdash_a a?\kappa_1 \,;\, \kappa'_1 =_D a?\kappa_2 \,;\, \kappa'_2}$$

## 8.3.3 Equivalence of systems

The following equivalence relation is useful in the proof of soundness of domain bisimilarity given in Theorem 8.26. The relation itself is too fine as a behavioural equivalence however as the systems it equates are identical in their plain form and only differ in provenance. The equivalence allows the two systems to use any domain equivalent provenance for the same value.

**Definition 8.18.** Let $D$ be a domain. Two $D$-systems $S$ and $T$ are said to be equivalent under environment $\Delta$, written $\Delta \vdash S =_D T$, if they do not contain any messages, their plain counterparts are the same, and their provenance sequences are $D$-equivalent under environment $\Delta$.

Lemma 8.19 states that systems related by the equivalence relation of Definition 8.18 remain equivalent after input actions.

**Lemma 8.19.** *Let $D$ be a domain, $\Delta$ be an environment, and $S$ and $T$ be two $D$-systems such that $\Delta \vdash S =_D T$. If $S \xrightarrow{m\langle p:\kappa\rangle}_f S'$, then for all $\kappa'$ such that $\Delta \vdash \kappa =_D \kappa'$, it holds that $T \xrightarrow{m\langle p:\kappa'\rangle}_f T'$, for some $T'$ such that $\Delta.(\kappa, \kappa') \vdash S' =_D T'$.*

*Proof.* Assume $S$ and $T$ are two $D$-systems such that $S =_D T$. Assume also that $S \xrightarrow{m\langle p:\kappa\rangle}_f S'$. We proceed by rule induction on the derivation of this transition.

The first case is FAct Rcv, which means that the transition has the following form:

$$\frac{j \in I \quad view(a, \kappa) \models \pi_j}{a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i] \xrightarrow{m\langle p:\kappa\rangle}_f a[P_j\{^{p:a?\kappa_m;\kappa}/_x\}]}$$

Since $S$ is of the form $a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i]$ and $S =_D T$, then $T$ must be of the form $a[\Sigma_{i\in I} m{:}\kappa'_m\,(\pi_i \text{ as } x).P'_i]$ where $\kappa'_m =_D \kappa_m$ and $P'_i =_D P_i$ for $i \in I$. Now, Let $\kappa'$ be a provenance sequence such that $\kappa =_D \kappa'$. We know that $a$ is in the domain $D$, which means that $view(a, \kappa') \models \pi_j$, and hence we should be able to derive the following transition:

$$\frac{j \in I \quad view(a, \kappa') \models \pi_j}{a[\Sigma_{i\in I} m{:}\kappa'_m\,(\pi_i \text{ as } x).P'_i] \xrightarrow{m\langle p:\kappa'\rangle}_f a[P'_j\{^{p:a?\kappa'_m;\kappa'}/_x\}]}$$

It is easy to see that since $P_j =_D P'_j$ and $p{:}a?\kappa_m \,;\, \kappa =_D p{:}a?\kappa'_m \,;\, \kappa'$, then it holds that $a[P_j\{^{p:a?\kappa_m;\kappa}/_x\}] =_D a[P'_j\{^{p:a?\kappa'_m;\kappa'}/_x\}]$.

There are several other cases, but they should all be straightforward. We illustrate FAct Exp and FAct Par-l below.

Case FAct Exp means that $S \xrightarrow{m\langle p:\kappa\rangle}_f S'$ has the form:

$$\frac{a[P] \xrightarrow{m\langle p:\kappa\rangle}_f S'' \quad n \notin n(m\langle p:\kappa\rangle)}{a[(\nu n)P] \xrightarrow{m\langle p:\kappa\rangle}_f (\nu n)S''}$$

We know that $T$ is of the form $a[(\nu n)P']$ such that $P =_D P'$, and by the induction hypothesis, we also know that $a[P'] \xrightarrow{m\langle p:\kappa'\rangle}_f T''$ where $S'' =_D T''$ and $\kappa =_D \kappa'$. We can then use FAct Exp to derive the transition $a[(\nu n)P'] \xrightarrow{m\langle p:\kappa'\rangle}_f (\nu n)T''$, and since $S'' =_D T''$ we can conclude that $(\nu n)S'' =_D (\nu n)T''$.

Case FAct Par-l means the transition $S \xrightarrow{m\langle p:\kappa\rangle}_f S'$ had the form:

$$\frac{S_1 \xrightarrow{m\langle p:\kappa\rangle}_f S_2 \quad bn(m\langle p:\kappa\rangle) \cap fn(R) = \emptyset}{S_1 \mid R \xrightarrow{m\langle p:\kappa\rangle}_f S_2 \mid R}$$

Again, $T$ must be of the form $T_1 \mid R$ such that $S_1 =_D T_1$, which implies by the induction hypothesis that $T_1 \xrightarrow{m\langle p:\kappa'\rangle}_f T_2$ such that $S_2 =_D T_2$ and $\kappa =_D \kappa'$. Now, we can use FAct Par-l to derive the transition $T_1 \mid R \xrightarrow{m\langle p:\kappa'\rangle}_f T_2 \mid R$, and since $S_2 =_D T_2$, we can conclude that $S_2 \mid R =_D T_2 \mid R$. The other cases are similar. □

With these definitions and results in place, we are now ready to define domain bisimilarity. Let us first explain the intuitions behind it. The aim of domain bisimilarity is to provide a coinductive characterisation of domain congruence. As such, domain bisimilarity must only equate systems if they are equivalent under domain congruence. We have seen that the labelled transition system is not observational, and therefore we cannot rely on the naive matching of action labels in the definition of bisimilarity. Instead, for each action label, we should consider the intended domain of the observer and equate the labels if the observer would not be able to discern any difference between them. More specifically, we have 4 cases:

- Internal actions $\tau$ contain no provenance and therefore the way they are perceived by the observer does not depend on its domain. Since we are interested in the weak version of the equivalence, a $\tau$ action from one system may be matched by any number of $\tau$ actions from the other.

- A free output action $\overline{m}\langle p : \kappa_1\rangle$ exports an annotated value $m : \kappa_1$ for the observer to consume on channel $m$. By listening on channel $m$ and receiving the value, the observer can identify the channel name of the output action. Name matching allows the observer to discern the plain value $p$ being sent. Finally, provenance checking enables the observer to detect any provenance included in the views of its principals. Therefore, the other system needs to match the output channel $m$, the plain value $p$ but not the entire provenance sequence $\kappa_1$. Instead, it can offer any provenance $\kappa_2$ as long as it ensures the views assigned to the observer are the same as those given by $\kappa_1$.

- A bound output action $\overline{m}(p : \kappa_1)$ is treated similarly to the free output action $\overline{m}\langle p : \kappa_1\rangle$ as described above.

- An input action $m\langle p : \kappa_1\rangle$ denotes that the system may receive an annotated value $p : \kappa_1$ from the observer on channel $m$. Since communication is asynchronous, the observer cannot detect directly if and when the value is actually consumed by the system. Therefore, input actions may be matched in two ways. Firstly, they can be matched by an equivalent input action $m\langle p : \kappa_2\rangle$. Like with output, the provenance of the matching input action $\kappa_2$ can be different from $\kappa_1$ as long as it matches the views given to principals belonging to the observer. It is important to note here that the two values $p : \kappa_1$ and $p : \kappa_2$ are meant to represent values sent to the observer in previous interactions. The observer may feed them back to the two systems under testing and therefore it is important for the two systems to ensure their reactions are the same, otherwise they would be divulging hidden provenance. Secondly, a system does not have to match the input actions of another system. It can opt to leave the message received from the observer unconsumed as long as it can match any future actions of the other system.

Domain bisimilarity is defined with respect to an equivalence environment $\Delta$. As we saw, equivalence environments are used to keep track of provenance sequences sent by the two systems. Every time an output action $\overline{m}\langle p{:}\kappa_1\rangle$ is performed by one system and a matching one $\overline{m}\langle p{:}\kappa_2\rangle$ is performed by the other, the two provenance sequences $(\kappa_1, \kappa_2)$ are added to the environment $\Delta$. Every time an input action $m\langle p{:}\kappa_1\rangle$ is performed by one system, the other system has to be able to match it for every provenance sequence $\kappa_2$ such that $\Delta \vdash \kappa_1 =_D \kappa_2$. This is to account for the possibility that the observer may send back two values it received from the two systems in a previous interaction.

In Definition 8.20, we formalise the notions of domain simulations, domain bisimulations and domain bisimilarities.

**Definition 8.20** (Domain bisimilarity)**.** Let $D$ be an environment, and let $\mathcal{S}$ be a relation containing triples of the form $(\Delta, S, T)$ where $\Delta$ is a $D$-environment and $S$ and $T$ are systems. We say that $\mathcal{S}$ is a $D$-simulation if, for all $(\Delta, S, T) \in \mathcal{S}$, it holds that:

- if $S \overset{\tau}{\longrightarrow}_f S'$ then $T \overset{\tau}{\longrightarrow}{}^*_f T'$ for some $T'$ such that $(\Delta, S', T') \in \mathcal{S}$.
- if $S \overset{\overline{m}\langle p:\kappa_1\rangle}{\longrightarrow}_f S'$ then $T \overset{\overline{m}\langle p:\kappa_2\rangle}{\Longrightarrow}_f T'$ for some $\kappa_2$ and $T'$ such that $\kappa_1 =_D \kappa_2$ and $(\Delta.(\kappa_1, \kappa_2), S', T') \in \mathcal{S}$.
- if $S \overset{\overline{m}(p:\kappa_1)}{\longrightarrow}_f S'$ then $T \overset{\overline{m}(p:\kappa_2)}{\Longrightarrow}_f T'$ for some $\kappa_2$ and $T'$ such that $\kappa_1 =_D \kappa_2$ and $(\Delta.(\kappa_1, \kappa_2), S', T') \in \mathcal{S}$.
- if $S \overset{m\langle p:\kappa_1\rangle}{\longrightarrow}_f S'$ then for all $\kappa_2$ such that $\Delta \vdash \kappa_1 =_D \kappa_2$, it holds that either:

– $T \overset{m\langle p{:}\kappa_2\rangle}{\Longrightarrow}_f T'$ for some $T'$ such that $(\Delta, S', T') \in \mathcal{S}$. Or,

– $T \overset{\tau}{\longrightarrow}^*_f T'$ for some $T'$ such that $(\Delta, S', T' \mid m\langle\langle p{:}\kappa_2\rangle\rangle) \in \mathcal{S}$.

If $\mathcal{S}$ is a simulation, then we use $\mathcal{S}^{-1}$ to denote its inverse, defined as the relation containing the triple $(\Delta, T, S)$ for every triple $(\Delta, S, T) \in \mathcal{S}$. We say that $\mathcal{S}$ is a $D$-bisimulation if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are $D$-simulations. We define $D$-bisimilarity, written $\approx_D$, to be the largest $D$-bisimulation.

The following lemmas prove some properties of domain bisimilarities. Lemma 8.21 states that any domain bisimilarity is an equivalence relation. Lemma 8.22 states that structurally congruent systems are equivalent under every domain bisimilarity. Lemmas 8.23 and 8.24 state that domain bisimilarities are preserved under restriction and parallel composition respectively. Together these two mean domain bisimilarities are contextual. This is stated in Lemma 8.25

**Lemma 8.21.** *Any domain bisimilarity $\approx_D$ is an equivalence relation.*

*Proof.* Let $D$ be a domain. We know that $\approx_D$ is symmetric by definition. To show that it is reflexive, consider the relation $r = \{(\Delta, S, S) \mid S \text{ is a system}\}$. It is easy to show that $r$ is a $D$-bisimulation by exhibiting that it satisfies the transfer property of Definition 8.11. The case for transitivity is similar. □

**Lemma 8.22.** *$S \equiv_f T$ implies $S \approx_D T$.*

*Proof.* We assume that $S \equiv_f T$ and proceed by induction on the derivation of this. The details are long but straightforward. □

**Lemma 8.23.** *$\Delta \vdash S \approx_D T$ implies $\Delta \vdash (\nu n)S \approx_D (\nu n)T$ for all channel names $n$.*

*Proof.* Let $r'$ be the relation containing all tuples of the form $(\Delta, (\nu n)S, (\nu n)T)$ such that $\Delta \vdash S \approx_D T$, and let $r$ be the union of $\approx_D$ and $r'$. We show that $r$ is a $D$-bisimulation up to structural congruence.

Assume that $(\Delta, S, T) \in r$. We need to prove that $(\Delta, S, T)$ has the transfer property of $D$-bisimulation, and since $r$ is clearly symmetric, we only need to prove this in one direction. We do this by induction on why $(\Delta, S, T)$ is in $r$. We have two cases: either $\Delta \vdash S \approx_D T$, in which case we already know that the transfer property holds, or $(\Delta, S, T)$ is of the form $(\Delta, (\nu n)S', (\nu n)T')$ for some name $n$ and systems $S'$ and $T'$ such that $\Delta \vdash S' \approx_D T'$. If the latter case, let us assume that $(\nu n)S'$ performs an action $\alpha_1$; we need to find a matching move by $(\nu n)T'$ as required by Definition 8.20. There are only two rules in the LTS of Figure 8.1 whose conclusion matches such a transition: FAct Res and FAct Open. In the case of FAct Res, this means that the derivation was of the form

$$\frac{S' \xrightarrow{\alpha_1}_f S''}{(\nu n)S' \xrightarrow{\alpha_1}_f (\nu n)S''}\ n \notin n(\alpha_1)$$

If $\alpha_1$ is not an input action, then $T' \overset{\alpha_2}{\Longrightarrow}_f T''$ for some $\alpha_2$ and $T''$ such that $\alpha_1 =_D \alpha_2$ and $\Delta.(\alpha_1, \alpha_2) \vdash S'' \approx_D T''$. From this, we can use FAct Res to get $(\nu n)T' \overset{\alpha_2}{\Longrightarrow}_f (\nu n)T''$, and since $\Delta.(\alpha_1, \alpha_2) \vdash S'' \approx_D T''$, we can conclude that $(\Delta.(\alpha_1, \alpha_2), (\nu n)S'', (\nu n)T'') \in r$. If $\alpha_1$ is an input action, i.e., of the form $m\langle p{:}\kappa_1\rangle$, then for all $\kappa_2$ such that $\Delta \vdash \kappa_1 =_D \kappa_2$, either $T' \overset{m\langle p:\kappa_2\rangle}{\Longrightarrow}_f T''$ such that $\Delta \vdash S'' \approx_D T''$, or $T' \xrightarrow{\tau}{}^{*}_f T''$ such that $\Delta \vdash S'' \approx_D T'' \mid m\langle\langle p{:}\kappa_2\rangle\rangle$. In the first case, we can use FAct Res to get $(\nu n)T' \overset{m\langle p:\kappa_2\rangle}{\Longrightarrow}_f (\nu n)T''$, and since $\Delta \vdash S'' \approx_D T''$, we conclude that $(\Delta, (\nu n)S'', (\nu n)T'') \in r$. In the second case, using FAct Res we get $(\nu n)T' \xrightarrow{\tau}{}^{*}_f (\nu n)T''$, and since $\Delta \vdash S'' \approx_D T'' \mid m\langle\langle p{:}\kappa_2\rangle\rangle$, we can infer that $(\Delta, (\nu n)S'', (\nu n)(T'' \mid m\langle\langle p{:}\kappa_2\rangle\rangle)) \in r$. This is all that is needed since $(\nu n)(T'' \mid m\langle\langle p{:}\kappa_2\rangle\rangle) \equiv_f (\nu n)T'' \mid m\langle\langle p{:}\kappa\rangle\rangle$. In the case of FAct Open, the derivation would be of the form:

$$\frac{S' \xrightarrow{\overline{m}\langle n:\kappa_1\rangle}_f S''}{(\nu n)S' \xrightarrow{\overline{m}(n:\kappa_1)}_f S''}\ n \neq m$$

Since $\Delta \vdash S' \approx_D T'$, this implies that $T' \overset{\overline{m}\langle n:\kappa_2\rangle}{\Longrightarrow}_f T''$ such that $\kappa_1 =_D \kappa_2$ and $\Delta.(\kappa_1, \kappa_2) \vdash S'' \approx_D T''$. Now using FAct Open, we can get $(\nu n)T' \overset{\overline{m}(n:\kappa_2)}{\Longrightarrow}_f T''$ and since we know that $\Delta.(\kappa_1, \kappa_2) \vdash S'' \approx_D T''$, we can conclude that $(\Delta.(\kappa_1, \kappa_2), S'', T'') \in r$. □

**Lemma 8.24.** $\Delta \vdash S \approx_D T$ *implies that* $\Delta \vdash S \mid R \approx_D T \mid R$ *for all initial D-systems R.*

*Proof.* We need to find a $D$-bisimulation that includes the triple $(\Delta, S \mid R, T \mid R)$. To do that, we define the relation $r$ to be the smallest relation satisfying the following:

| | |
|---|---|
| $(\Delta, S, T) \in r$ | if $\Delta \vdash S \approx_D T$ |
| $(\Delta, (\nu n)S, (\nu n)T) \in r$ | if $(\Delta, S, T) \in r$ |
| $(\Delta, S \mid R_1, T \mid R_2) \in r$ | if $(\Delta, S, T) \in r$ and $\Delta \vdash R_1 =_D R_2$. |

Note the last clause of the definition; this is a result of our transfer property being weaker than what is traditionally used in the definition of bisimulation (output provenance is only required to match up to the view of principals in $D$). The need for this will become clearer when we prove the cases FAct Comm-l and FAct Close-l below. The pair $(S \mid R, T \mid R)$, what is actually required for closure under parallel composition, is simply a special case of $(S \mid R_1, T \mid R_2)$ in the above definition since $=_D$ is reflexive.

We need to show that $r$ is a $D$-bisimulation up-to $\equiv_f$. Again, it is clear that $r$ is symmetric in the last two elements and so we only need to show the transfer property in one direction. We do this by induction on why $(\Delta, S, T)$ is in the relation $r$. There are three cases.

In the first case, $\Delta \vdash S \approx_D T$ and therefore the transfer property holds since $\approx_D$ is a $D$-bisimulation.

In the second case, $(\Delta, S, T)$ is of the form $(\Delta, (\nu n)S', (\nu n)T')$ for some $S'$ and $T'$ such that $(\Delta, S', T') \in r$. We assume that $(\nu n)S'$ performs an action $\alpha_1$; we then need to find a matching move by $(\nu n)T'$ as required by the definition of bisimulation. The proof of this is similar to that of Lemma 8.23.

In the third and final case, $(\Delta, S, T)$ is of the form $(\Delta, S' \mid R_1, T' \mid R_2)$ where $(\Delta, S', T') \in r$ and $R_1$ and $R_2$ are $D$-systems such that $\Delta \vdash R_1 =_D R_2$. Let us assume that $S' \mid R_1$ performs an action $\alpha_1$; we need to find a matching move by $T' \mid R_2$ as required by the definition of $D$-bisimulation. There are 6 rules that could have been used to derive the transition of $S' \mid R_1$: FAct Par-l, FAct Comm-l and FAct Close-l, as well as their symmetric counterparts.
In the case of FAct Par-l, the derivation would be of the form:

$$\frac{S' \xrightarrow{\alpha_1}_f S'' \quad bn(\alpha_1) \cap fn(R_1) = \emptyset}{S' \mid R_1 \xrightarrow{\alpha_1}_f S'' \mid R_1}$$

If $\alpha_1$ is not an input action, then by the induction hypothesis, $T' \overset{\alpha_2}{\Longrightarrow}_f T''$ such that $\alpha_1 =_D \alpha_2$ and $(\Delta.(\alpha_1, \alpha_2), S'', T'') \in r$. Using FAct Par-l, we can derive the transition $T' \mid R_2 \overset{\alpha_2}{\Longrightarrow}_f T'' \mid R_2$. Since $(\Delta.(\alpha_1, \alpha_2), S'', T'') \in r$ and $\Delta \vdash R_1 =_D R_2$, we can conclude that $(\Delta.(\alpha_1, \alpha_2), S'' \mid R_1, T'' \mid R_2) \in r$. If $\alpha_1$ is an input action, that is, of the form $m\langle p{:}\kappa_1\rangle$, then the induction hypothesis implies that for all $\kappa_2$ such that $\Delta \vdash \kappa_1 =_D \kappa_2$, either $T' \overset{m\langle p:\kappa_2\rangle}{\Longrightarrow}_f T''$ such that $(\Delta, S'', T'') \in r$, or $T' \xrightarrow{\tau}{}^{*}_f T''$ such that $(\Delta, S'', T'' \mid m\langle\langle p{:}\kappa_2\rangle\rangle) \in r$. In the first case, we can use FAct Par-l to get $T' \mid R_2 \overset{m\langle p:\kappa_1\rangle}{\Longrightarrow}_f T'' \mid R_2$ where since we know that $(\Delta, S'', T'') \in r$ and $\Delta \vdash R_1 =_D R_2$, we can conclude that $(\Delta, S'' \mid R_1, T'' \mid R_2) \in r$. In the second case, using FAct Par-l, we can derive the transition $T' \mid R_2 \xrightarrow{\tau}{}^{*}_f T'' \mid R_2$. Now, from $(\Delta, S'', T'' \mid m\langle\langle p{:}\kappa_1\rangle\rangle) \in r$ we can conclude that $(\Delta, S'' \mid R_1, T'' \mid m\langle\langle p{:}\kappa_1\rangle\rangle \mid R_2) \in r$, which is what we need since $T'' \mid m\langle\langle p{:}\kappa_1\rangle\rangle \mid R_2 \equiv_f T'' \mid R_2 \mid m\langle\langle p{:}\kappa_1\rangle\rangle$.
In the case of FAct Comm-l, the transition would take the form:

$$\frac{S' \xrightarrow{\overline{m}\langle p:\kappa_1\rangle}_f S'' \quad R_1 \xrightarrow{m\langle p:\kappa_1\rangle}_f R_1'}{S' \mid R_1 \xrightarrow{\tau}_f S'' \mid R_1'}$$

From $S' \xrightarrow{\overline{m}\langle p:\kappa_1\rangle}_f S''$, the induction hypothesis implies $T' \overset{\overline{m}\langle p:\kappa_2\rangle}{\Longrightarrow}_f T''$ such that $\kappa_1 =_D \kappa_2$ and $(\Delta.(\kappa_1,\kappa_2), S'', T'') \in r$. In addition to this, by Lemma 8.19, $R_1 \xrightarrow{m\langle p:\kappa_1\rangle}_f R_1'$ and $\kappa_1 =_D \kappa_2$ imply $R_2 \xrightarrow{m\langle p:\kappa_2\rangle}_f R_2'$ such that $\Delta.(\kappa_1,\kappa_2) \vdash R_1' =_D R_2'$. Using FAct Comm-l, we are able to derive the transition $T' \mid R_2 \xrightarrow{\tau}{}^*_f T'' \mid R_2'$. And, since we have $(\Delta.(\kappa_1,\kappa_2), S'', T'') \in r$ and $\Delta.(\kappa_1,\kappa_2) \vdash R_1' =_D R_2'$, we can conclude that $(\Delta.(\kappa_1,\kappa_2), S'' \mid R_1', T'' \mid R_2') \in r$. The case of FAct Close-l proceeds in a similar fashion but with bound output actions.
In the case of FAct Par-r, the transition would take the following form:

$$\frac{R_1 \xrightarrow{\alpha_1}_f R_1' \quad bn(\alpha_1) \cap fn(S') = \emptyset}{S' \mid R_1 \xrightarrow{\alpha_1}_f S' \mid R_1'}$$

Now, since $R_1 =_D R_2$ (by assumption) and $=_D \subseteq \approx_D$ , we can derive the matching move for $R_2$ and combine it with $T'$ using FAct Par-r to get the required move for $T' \mid R_2$.
In the case of FAct Comm-r, the transition would take the following form:

$$\frac{S' \xrightarrow{m\langle p:\kappa_1\rangle}_f S'' \qquad R_1 \xrightarrow{\overline{m}\langle p:\kappa_1\rangle}_f R_1'}{S' \mid R_1 \xrightarrow{\tau}_f S'' \mid R_1'}$$

The induction hypothesis implies that either $T' \overset{m\langle p:\kappa_1\rangle}{\Longrightarrow}_f T''$ such that $(S'', T'') \in r$, or $T' \xrightarrow{\tau}{}^*_f T''$ such that $(S'', T'' \mid m\langle\langle p:\kappa_1\rangle\rangle) \in r$. $R_1 \xrightarrow{\overline{m}\langle p:\kappa_1\rangle}_f R_1'$ implies that $R_2 \xrightarrow{\overline{m}\langle p:\kappa_2\rangle}_f R_2'$ such that $\kappa_1 =_D \kappa_2$. □

**Lemma 8.25.** *$S \approx_D T$ implies $C[S] \approx_D C[T]$ for all D-contexts C.*

*Proof.* We know from Lemma 8.23 that $\approx_D$ is preserved by restriction. By Lemma 8.24, and a symmetric version of it, we know that $\approx_D$ is preserved by parallel composition with $D$-systems. Hence, we conclude that $\approx_D$ is preserved by all $D$-contexts. □

**Theorem 8.26.** *$S \approx_D T$ implies $S \cong_D T$.*

*Proof.* We need to show that $\approx_D$ satisfies all the defining properties of $\cong_D$, that is, it is symmetric, it preserves observables, it is closed under reduction and that it is D-contextual. From that, and the fact that $\cong_D$ is by definition the largest relation satisfying those conditions, it will follow that $\approx_D \subseteq \cong_D$, i.e., $S \approx_D T$ implies $S \cong_D T$.

We know that $\approx_D$ is symmetric by definition.

To prove that $\approx_D$ preserves observables, let us assume that $S \approx_D T$ and $S \downarrow_f m$. This means that $S \equiv_f (\nu n)(m\langle\langle p:\kappa_1\rangle\rangle \mid S')$ and $n \neq m$. Using the rules of Figure 8.1, $S$ can make a free or bound output action depending on what $p$ is. If $p \neq n$, then $S$ can make a free output action as follows $S \xrightarrow{\overline{m}\langle p:\kappa_1\rangle}_f (\nu n)(0 \mid S')$. This implies, since $S \approx_D T$, that $T \overset{\overline{m}\langle p:\kappa_2\rangle}{\Longrightarrow}_f T'$ for some $\kappa_2$ and $T'$ such that $\kappa_1 =_D \kappa_2$ and $(\nu n)(0 \mid S') \approx_D T'$. Lemma 8.3 then implies that $T \Longrightarrow_f T''$ for some $T''$ such that $T'' \equiv_f m\langle\!\langle p:\kappa_2\rangle\!\rangle \mid T'$, which means that $T \Downarrow_f m$. If $p = n$, then $S$ can make a bound output action, and the proof proceeds in a similar fashion to the previous case but we end up with $T'' \equiv_f (\nu n)(m\langle\!\langle n:\kappa_2\rangle\!\rangle \mid T')$, which again means that $T \Downarrow_f m$.

To prove that $\approx_D$ is reduction closed, let us assume that $S \approx_D T$ and $S \longrightarrow_f S'$. By Lemma 8.4, this implies that $S \xrightarrow{\tau}_f S''$ and $S'' \equiv_f S'$. This in turn implies that $T \xrightarrow{\tau}{}^*_f T'$ and $S'' \approx_D T'$, which by Lemma 8.2 implies that $T \Longrightarrow_f T'$. Since $\approx_D$ is an equivalence relation and $\equiv_f$ is a subset of $\approx_D$ by lemmas 8.21 and 8.22 respectively, we conclude that $S' \approx_D T'$.

We already know by Lemma 8.25 that $\approx_D$ is $D$-contextual. Hence, we conclude that $\approx_D \subseteq \cong_D$. □

## 8.4 Concluding remarks

In this chapter, we strengthened the results of the previous chapter by studying behavioural equivalences for the provenance calculus with filters. The contextual equivalence we defined, domain congruence, provided a novel way of specifying properties of provenance disclosure policies. It enabled us to vary the privileges of the observer and study the impact of this on the discriminating power of the equivalence. This allowed us to provide a relative measure of the effectiveness of filters in hiding sensitive provenance information and implementing the provenance disclosure policies of principals. Note that we are using the term "relative measure" to refer to the ordering of filters in terms of how much provenance they hide from principals. To see how this works, consider that the aim of a filter is to specify which parts of a provenance sequence each principal is allowed to see. This is done by mapping group expressions to filtering expressions. Group expressions specify sets of principals and filtering expressions specify the parts of a provenance sequence that need to be hidden from the set of principals in question. Let us consider the following filter as an example:

$$\phi = G \mapsto_{\mathsf{over}} \mathit{hidemysources}_\Phi \;;\; (\sim - G) \mapsto_{\mathsf{over}} \mathit{id}_\Phi$$

This filter uses the filtering expression $\mathit{hidemysources}_\Phi$ to hide from principals in group $G$ the previous provenance of the value (before it reached the current principal) and the provenance of the channels used to receive it and send it. The filter leaves the provenance view of everyone else intact by associating the identity filtering expression $id_\Phi$ with the group expression $\sim - G$. There are two ways in which the filter $\phi$ could hide more provenance; either by hiding provenance from more principals or by hiding more provenance from the same principals. More concretely, consider the following two filters:

$$\phi' = (G + a) \mapsto_{\mathsf{over}} \mathit{hidemysources}_\Phi \,;\, (\sim - (G + a)) \mapsto_{\mathsf{over}} id_\Phi$$
$$\phi'' = G \mapsto_{\mathsf{over}} \mathit{off}_\Phi \,;\, (\sim - G) \mapsto_{\mathsf{over}} id_\Phi$$

Filter $\phi'$ differs from $\phi$ in that it associates $\mathit{hidemysources}_\Phi$ with group $G + a$ instead of $G$. This means that $\phi'$ hides the previous provenance of the value and the provenance of the channels used from a larger set of principals compared to filter $\phi$. Therefore, it could be said that filter $\phi'$ hides more provenance than filter $\phi$. Now consider the second filter $\phi''$, it associates the filter expression $\mathit{off}_\Phi$ with the same group expression $G$. So in terms of group expressions, it is similar to filter $\phi$. However, $\phi''$ hides the entire provenance sequence from principals in group $G$. Therefore, like $\phi'$, $\phi''$ too hides more provenance than $\phi$. Filters $\phi'$ and $\phi''$ are not as easily comparable though since they use different group expressions and different filtering expressions. This means that such comparisons of filters should induce a partial order between filters.

Domain congruence is a simple equivalence, at least as far as its definition is concerned. Trying to characterise it coinductively, however, is not an easy task as we saw with the definition of domain bisimilarity. The main source of the difficulty lies in the complexity of accounting for the exact provenance that an observer is able to discern. This is a non-trivial problem as we saw. The reason is that even when the views of provenance that are assigned to the observer prevent it from distinguishing between two provenance sequences, the observer may still tell them apart by watching for any information leakage in its future interactions with the systems. We proved that domain bisimilarity provides a sound proof method to show the equivalence of systems and left it as an open problem whether a complete proof method exists. Domain bisimilarity is interesting, not only as a tractable proof methodology for domain congruence, but also as a formal way of highlighting many of the subtleties underlying the interplay between provenance checking and filters. It is this interplay that made domain bisimilarity much more complicated to define than the bisimilarities of the plain and annotated versions of the calculus.

# Chapter 9

# Conclusion

Computer Science, being a science of the artificial, may be seen as serving two distinct but related purposes. On the one hand, it deals with how computing systems *are*, how they are built and what properties they may have. On the other hand, it is concerned with how computing systems *ought to be*, how they should be built to be more efficient, more robust and more dependable. This means that the subject matter of Computer Science is in fact a moving target. As our body of computing knowledge grows, we are able to build larger, more complex systems, and so far, we have never passed up the opportunity to do so. Computer Science is in a constant race to make the increasingly complex simpler.

Ubiquitous computing promises to bring about computer systems that are larger and far more complex than anything we have ever built before. These systems are predicted to be autonomous, mobile, and context-aware. They are predicted to be interconnected, communicating large amounts of data with each other. The data these systems communicate would be constructed from data they authored themselves as well as data they received from other systems. The heterogeneous nature of these systems would mean they have widely varying goals, quality standards, and trust policies. When a system depends on the data it receives from other systems, it would need to ensure this data is of a certain quality. One way to achieve this, which we think would be particularly suitable for such environments, is to track the provenance of data and use it to judge its quality.

We assume that principals in systems similar to those envisioned by ubiquitous computing would form webs of trust. These would reflect the trustworthiness that each principal ascribes to other principals it interacts with, and would naturally evolve over time in accordance with any new information that the principal learns from its interactions. For provenance to be useful as an indicator of data quality, it needs to associate with each value a record documenting where the value originated and the principals that contributed to its journey to its current state. This information needs to be kept up-to-date and needs to be made available to principals to allow them to make informed decisions about what data they wish to consume.

Provenance reveals the actions performed by a principal as they relate to a given value, and therefore the principal should have the ability to decide who that information is made available to, and in extreme cases to decide to withhold it from everyone, effectively opting out of the provenance tracking activity. Principals should still be able to detect when provenance information has been withheld from them to ensure any decisions they make based on provenance are not ill-informed.

## 9.1 Review of contributions

Our aim, as described and motivated in Chapter 1, is to study provenance-based trust in distributed systems. To that end, we presented the provenance calculus in Chapter 3. The calculus implements an intuitive notion of provenance that is intended as an account of all past events that led a piece of data to be in its current "state". Given the trust policy of a principal, the provenance of a piece of data then serves as an indicator of its quality as perceived by this principal. Concretely, the calculus annotates or tags each value with a sequence of events denoting its provenance. This is updated with new and relevant information dynamically as expressed by the provenance tracking semantics of the calculus. Principals make use of the provenance information through pattern matching which enables them to decide if and how to use the data. The details of pattern matching itself were orthogonal to our objectives however, and hence the calculus was made parametric on the choice of a pattern matching language. In Section 3.3, we presented a sample pattern language and gave several examples to illustrate the use of the calculus. Our contributions here are threefold; a notion of provenance for distributed systems, a run time provenance tracking method and an integrated provenance checking mechanism. In fact, to the best of our knowledge, this is the first work that incorporates provenance management into a process calculus. We think this is worthwhile since it provides a unified framework to study all the elements of provenance management. Not only that, but it also brings a rich set of tools and techniques from the concurrency community to the study of provenance.

In Chapter 4, we highlighted the role of provenance tracking in the calculus by comparing the annotated version of the calculus to a plain one in which no provenance tracking or checking take place. We strengthened these results further by studying behavioural equivalences for the annotated version and contrasting them with their counterparts in the plain version. We believe behavioural equivalences provide a useful and novel way to study the properties of provenance. In particular, by analysing the extra discriminating power that a behavioural equivalence gains as a result of provenance, we are able to better highlight the role of provenance management in the calculus.

In Chapter 5, we studied some properties of provenance. The first property we looked at, well-formedness, allowed us to separate semantically meaningful provenance sequences from syntactically legal ones. We also proposed a denotational semantics for provenance sequences in the form of formulae in a temporal logic with a past modality. We used this to define a notion of provenance correctness and proved that the provenance tracking semantics of the calculus preserves the correctness of provenance. These results are important since they formalise the guarantees that we can expect from the provenance calculus. Moreover, we believe the definition of the past logic and its use to define the semantics of provenance sequences constitute an original approach for understanding provenance and its meaning as a concrete record of the past. In light of this, system transitions provide the models against which the truth of provenance may be judged.

In Chapter 6, we discussed static provenance tracking and proposed a static type and effect system to accomplish this. The type and effect system guaranteed that principals always receive data with provenance that matches their provenance policies without the run time overhead incurred as a result of dynamic provenance checks. In addition to providing an alternative method to track the provenance of values and enforce the provenance policies of principals, the type and effect system offered an interesting view of provenance, provenance annotations as types. Unlike traditional type systems, the provenance type system deals with dynamic behavioural types. This, we hope, provides interesting results from both the provenance point of view as well as the typing point of view.

In Chapter 7, we looked at the problem of provenance security. Our primary aim was to address the privacy and security concerns that arise from the disclosure of provenance. We separated the problem into two parts, what aspects of provenance need to be protected and how to extend the calculus to achieve that. This led us to define the two notions of weakenings and filters. For the benefit of keeping our results generic, we provided a specification for filter languages and left the details of filters unspecified. We defined a sample filter language to illustrate our ideas however. This language employed a novel notion of modes that allowed us to achieve full dynamic change of provenance visibility without violating the security policies of principals.

Chapter 8 studied domain congruence, a behavioural equivalence that is sensitive to the set of principals available to the observer. The aim of this was to enable us to reason about the security guarantees offered by filters. We also proposed a sound proof method for domain congruence based on bisimulation. The definition of this latter highlighted many of the subtle properties of filters and their interaction with provenance checking. It also demonstrated the usefulness of concurrency theory tools such as bisimulation in studying provenance management and its properties.

## 9.2 Possibilities for future work

In this section, we propose several avenues for future work. These range from simple extensions to the calculus to more elaborate efforts aimed at generalising our provenance notion, investigating other notions as well as considering possible implementation strategies for various aspects of the work presented in this thesis.

### 9.2.1 General extensions to the calculus

The main aim of these extensions is to make it easier to express certain systems that require features not currently available in the calculus and to reason about their properties. As we already commented in Section 3.3 when presenting our sample pattern matching language, our patterns are closer to types than they are to patterns as commonly found in other programming contexts. Indeed, pattern matching usually allows the programmer to deconstruct terms by binding against their subparts. In our pattern language, we did not want to allow this as it would have introduced provenance as data which would have overcomplicated the calculus and the provenance notion. One may wish, however, to extend the pattern language with bindings and study the implications this has on provenance tracking, especially in a setting where the calculus is extended with data terms.

Such an extension requires the definition of pattern matching languages to be modified so that patterns include variables which may be bound to parts of the provenance sequence when performing pattern matching. Technically, this involves adding a function that returns the free variables of a pattern and modifying the pattern matching relation so that it returns a substitution when a successful pattern match takes place. The new definition of pattern matching languages would be as follows.

**Definition 9.1.** A pattern matching language is a triple $(\Pi, fv, \models)$ where $\Pi$ is a set of patterns ranged over by $\pi, \pi', \ldots$, $fv : \Pi \to \mathcal{P}(\mathcal{X})$ is a function that returns the free variables in a pattern and $\models\ \subseteq \mathcal{K} \times \Pi \times \Sigma$, called the pattern matching relation, relates provenance sequences to patterns and substitutions.

In this new setting, the receive rule would also need to be modified to something like:

$$\frac{\kappa_p \models (\pi_j, \sigma) \qquad j \in I}{a[\Sigma_{i\in I} m\,(\pi_i \;\mathsf{as}\; x).P_i] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \to_a a[P_j \sigma' \{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

where $\sigma$ is the substitution resulting from the pattern match of $\kappa_p$ to $\pi_j$ and $\sigma'$ is like $\sigma$ but with the values annotated with provenance that reflects their origin.

In addition to this, we might also like to extend patterns to include "normal" variables so that dynamic information such as a newly learned principal names may be used in pattern matching.

We may also consider an extension to the calculus that allows the dynamic creation of principal names. This might have the following syntax

$$\mathsf{newp}\; x \;\mathsf{in}\; P$$

with its semantics given by the following rule

$$a[\mathsf{newp}\; x \;\mathsf{in}\; P] \to_a (\nu b)(b[P\{^{b:a}/_x\}])$$

The above rule allows a new principal name $b$ to be created and migrates the process $P$ to it. Note the restriction of the newly created name to denote that it is a fresh name and the substitution of $b : a$ for the occurrences of $x$ to denote that the new identity $b$ was created by $a$. This ensures that any principal that receives data from this new principal does so knowing that $b$, having been created by $a$, is simply an alias of $a$ and therefore should be assigned the same trustworthiness level as $a$. This guarantees that principals cannot cheat and fool other principals into accepting data from them by creating new identities. To keep it consistent with principal name creation, channel name creation might also be modified to use the syntax

$$\mathsf{newc}\; x \;\mathsf{in}\; P$$

and the semantics

$$a[\mathsf{newc}\; x \;\mathsf{in}\; P] \to_a (\nu m)(a[P\{^{m:a}/_x\}])$$

which creates a new channel name $m$ that is private to principal $a$. Here too, the substitution of $m : a$ for $x$ is used to denote that the channel name $m$ was created by $a$.

### 9.2.2 Other provenance notions

The notion of provenance to employ in a particular setting depends on both the subject of provenance and its purpose. Here, the subject of provenance refers to the entity about which provenance information is recorded, which in our case for example happens to be messages exchanged between principals. The subject of provenance determines what type of information is available and meaningful to keep. On the other hand, the purpose of provenance tracking determines how much information exactly one needs to record. In our case, provenance is kept to allow principals to use their local beliefs about the trustworthiness of other principals to decide which data to consume. This meant that information about the role each principal played in getting a piece of data to its current state was crucial. Implied in this was also the order and the number of times a particular principal affected a value. Our representation of provenance as a sequence of events does indeed reflect this fact.

It would be interesting to investigate other notions of provenance, starting with those in line with the motivations and setting of this work, and later moving to consider other settings that might be of interest such as that of higher order calculi for example. In this latter, one could envisage recording information about both the role different principals played in affecting the migration of a process as well as its execution history at different sites. Considering our notion of provenance, we may see it as defined by three elements: the order of events, their multiplicity and what actions are considered to affect a value. A change to any of these would lead to a different, albeit related, provenance notion. For example, one could consider a provenance notion where order and multiplicity are unimportant and where only the identities of principals are recorded and not the provenance of the channels used. A straightforward representation of provenance for such a notion would be as a set of principal names and its semantics would be given by modifying the send and receive rules as follows:

$$\textsc{ARed Snd}\quad \frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \rightarrow_a m\langle\!\langle p{:}\kappa_p \cup \{a\}\rangle\!\rangle}$$

$$\textsc{ARed Rcv}\quad \frac{\kappa_p \models \pi_j \quad j \in I}{a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i\ \mathsf{as}\ x).P_i] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \rightarrow_a a[P_j\{^{p:\kappa_p}/_x\}]}$$

where the pattern match $\kappa_p \models \pi_j$ is simply the set inclusion test $\kappa_p \subseteq \pi_j$.

One could generate many provenance notions by including or excluding order and multiplicity and varying the definition of what actions (and what parts of those actions) affect a value. In the calculus of Chapter 3, we considered send and receive actions to be important as they are what determines where a value is going to end up. From those, we only recorded the identities of principals involved and ignored the names of the channels used. Our justification there was that trust is a relation between principals and hence the name of the channel used for communication is not important. One could argue however that channel names need to be recorded as they tell a principal why a value was transmitted along the path it did. Moreover, if one considers name matching, as in the following example:

$$a[\mathsf{if}\ m{:}\kappa_m = n{:}\kappa_n\ \mathsf{then}\ l\,\langle v\rangle\ \mathsf{else}\ l\,\langle v'\rangle\ ] \mid b[l\,(x).P]$$

one sees that the value principal $b$ ends up with depends the outcome of the equality test. Hence, the "if-action" (together with the values tested and their provenances) should be recorded in the provenance of the value sent by $a$ ($v'$ in this case since the two names tested are not equal). A similar argument may be raised regarding input prefixes as their successful execution determines whether any message in the continuation gets sent or not.

The different provenance notions induced by these changes are related and some of them are contained in others. That is, the information recorded in a particular provenance notion may be completely contained in that of another provenance notion. Such is the case between the set based notion we described above and our sequence based provenance notion for example.

### 9.2.3 Implementation concerns

There are a number of implementation concerns worth investigating. The first has to do with the implementability of our provenance notion itself which requires strong integrity and non-repudiation guarantees (formulated as the provenance correctness property in Chapter 5). Such a guarantee could be fairly easily provided by a trusted middleware layer. However, in the absence of such an environment, ensuring the integrity and non-repudiation of provenance information seems to be more difficult. A simpler provenance notion than ours, one where only the identities of principals along the path of a value are kept, could be implemented using a digital signing scheme. The notion of provenance described in this work, however, also requires the provenance of channels (and the type of action) to be recorded which makes a simple application of digital signatures impossible. Furthermore, the security requirements demanded by filters mean that any implementation strategy is not going to be straightforward.

## 9.3 Summary and concluding remarks

This thesis introduced an extension of the asynchronous $\pi$-calculus aimed at studying provenance-based trust in data quality. The extension consisted of annotating all exchangeable data terms with metadata denoting their provenance. In the standard semantics of the calculus, the provenance information was automatically updated as the system evolved and the trust policies of principals were checked and enforced dynamically. Behavioural equivalences for both the plain and the annotated versions of the calculus were studied, highlighting the differences between the two versions and making clear the role of provenance tracking in the calculus. Two properties of provenance, well-formedness and correctness were defined and studied. In addition to this, an alternative semantics relying on a static type and effect inference system was also proposed. This latter ensured that principals would always receive data with provenance that complies with their policies without the run time overhead of dynamic provenance tracking and checking. To address the security ramifications of provenance tracking, an extension of the calculus with filters was proposed. This enabled principals to control the disclosure of provenance information to other principals. Notions of behavioural equivalence were defined for this extended calculus and used to study the security properties of provenance. Finally, the contributions of the thesis were reviewed and avenues for future work were proposed.

# Appendix A

# Labelled Transition System Semantics for the Provenance Calculus

## A.1 Plain version

### A.1.1 Asynchronous bisimilarity

The definition of reduction barbed congruence involves a universal quantification over contexts, which renders it unsuitable for proving the equivalence of systems in general. To obtain a tractable proof method, we define asynchronous bisimilarity. The definition of this latter relies on the labelled transition system of Figure A.1, which provides an alternative formulation of the semantics of the plain version of the calculus.

The labelled semantics is given in terms of a family of binary relations on plain systems of the form $\xrightarrow{\alpha}_p$, where $\alpha$ ranges over actions. We have four kinds of actions as follows:

1. $\overline{m}\langle v \rangle$: which denotes the output of the *free* value $v$ on channel $m$.
2. $\overline{m}(n)$: which denotes the output of the *bound* name $n$ on channel $m$.
3. $m\langle v \rangle$: which denotes the input of the *free* value $v$ on channel $m$.
4. $\tau$: which denotes the unobservable (internal) action.

As indicated above, the name $n$ in action $\overline{m}(n)$ is bound; all other occurrences of names in actions are free. We use $bn(\alpha)$ and $fn(\alpha)$ for the sets of bound names and free names in action $\alpha$ respectively. We denote the union of these two by $n(\alpha)$.

**Definition A.1.** The family of plain transition relations $\xrightarrow{\alpha}_p$, where $\alpha$ ranges over actions, is defined by the rules of Figure A.1.

Figure A.1. *Plain semantics: labelled transition system*

PAct Snd
$$\frac{}{a[m\langle v\rangle] \xrightarrow{\tau}_p m\langle\langle v\rangle\rangle}$$

PAct Msg
$$\frac{}{m\langle\langle v\rangle\rangle \xrightarrow{\overline{m}\langle v\rangle}_p 0}$$

PAct Asc
$$\frac{}{0 \xrightarrow{\overline{m}\langle v\rangle}_p m\langle\langle v\rangle\rangle}$$

PAct Rcv
$$\frac{j \in I}{a[\Sigma_{i\in I} m(x).P_i] \xrightarrow{m\langle v\rangle}_p a[P_j\{^v/_x\}]}$$

PAct If$_t$
$$\frac{}{a[\text{if } m = m \text{ then } P \text{ else } Q\ ] \xrightarrow{\tau}_p a[P]}$$

PAct If$_f$
$$\frac{m \neq n}{a[\text{if } m = n \text{ then } P \text{ else } Q\ ] \xrightarrow{\tau}_p a[Q]}$$

PAct Res
$$\frac{S \xrightarrow{\alpha}_p S' \quad n \notin n(\alpha)}{(\nu n)S \xrightarrow{\alpha}_p (\nu n)S'}$$

PAct Open
$$\frac{S \xrightarrow{\overline{m}\langle n\rangle}_p S'}{(\nu n)S \xrightarrow{\overline{m}(n)}_p S'}$$

PAct Par
$$\frac{S \xrightarrow{\alpha}_p S' \quad bn(\alpha) \cap fn(T) = \emptyset}{S \mid T \xrightarrow{\alpha}_p S' \mid T}$$

PAct Comm
$$\frac{S \xrightarrow{\overline{m}\langle v\rangle}_p S' \quad T \xrightarrow{m\langle v\rangle}_p T'}{S \mid T \xrightarrow{\tau}_p S' \mid T'}$$

PAct Close
$$\frac{S \xrightarrow{\overline{m}(n)}_p S' \quad T \xrightarrow{m\langle n\rangle}_p T'}{S \mid T \xrightarrow{\tau}_p (\nu n)(S' \mid T')}$$

The labelled transition system provides an *alternative* formulation of the semantics of the plain version, and hence we expect it to coincide with the reduction semantics defined in Section 3.1.2. This is shown formally in Corollary A.6. To prove this latter as well as other results below, we need to first prove a number of lemmas concerning the labelled transition system and its relation to the reduction semantics. We do this below.

**Lemma A.2.** *$S \equiv_p T$ and $S \xrightarrow{\alpha}_p S'$ implies that $T \xrightarrow{\alpha}_p T'$ for some $T'$ such that $S' \equiv_p T'$.*

**Lemma A.3.** *$S \xrightarrow{\tau}_p S'$ implies that $S \to_p S'$.*

**Lemma A.4.** *The labelled semantics satisfies the following properties:*

- *$S \xrightarrow{\overline{m}\langle v\rangle}_p S'$ implies that $S \equiv_p m\langle\langle v\rangle\rangle \mid S'$.*

- $S \xrightarrow{\overline{m}(v)}_p S'$ *implies that* $S \equiv_p (\nu n)(m\langle\!\langle v \rangle\!\rangle \mid S')$.

**Lemma A.5.** $S \rightarrow_p S'$ *implies that* $S \xrightarrow{\tau}_p S''$ *for some* $S''$ *such that* $S' \equiv_p S''$.

**Corollary A.6.** $S \rightarrow_p S'$ *if and only if* $S \xrightarrow{\tau}_p S''$ *and* $S'' \equiv_p S'$.

*Proof.* Follows from Lemma A.3 and Lemma A.5. □

**Definition A.7** (Asynchronous bisimilarity)**.** (Plain, weak) asynchronous bisimilarity, $\approx_p$, is the largest symmetric binary relation on systems such that whenever $S \approx_p T$, it holds that if $S \xrightarrow{\alpha}_p S'$ and $bn(\alpha)$ are fresh, then $T \xrightarrow{\alpha}_p T'$ for some $T'$ such that $S' \approx_p T'$.

**Lemma A.8.** $S \equiv_p T$ *implies* $S \approx_p T$.

**Theorem A.9.** $S \approx_p T$ *implies* $S \cong_p T$.

*Proof.* We need to show that $\approx_p$ satisfies the three defining criteria of $\cong_p$, that is, preservation of barbs, closure under reduction and contextuality. From that, and the fact that $\cong_p$ is by definition the largest relation satisfying those three conditions, it will follow that $\approx_p \subseteq \cong_p$, i.e., $S \approx_p T$ implies $S \cong_p T$. □

**Theorem A.10.** $S \cong_p T$ *implies* $S \approx_p T$.

**Corollary A.11.** $\cong_p$ *and* $\approx_p$ *coincide.*

*Proof.* The proof follows directly from Theorem A.9 and Theorem A.10. □

## A.2 Annotated version

### A.2.1 Asynchronous bisimilarity

**Definition A.12.** The family of annotated transition relations $\xrightarrow{\alpha}_a$, where $\alpha$ ranges over actions, is defined by the rules of Figure A.3.

Figure A.2. *Annotated semantics: labelled transition system*

AAct Snd

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p\rangle] \xrightarrow{\tau}_a m\langle\!\langle p : a!\kappa_m \,;\, \kappa_p \rangle\!\rangle}$$

AAct Msg

$$\frac{}{m\langle\!\langle p : \kappa \rangle\!\rangle \xrightarrow{\overline{m}\langle p:\kappa\rangle}_a 0}$$

AAct Asc

$$\frac{}{0 \xrightarrow{\overline{m}\langle p:\kappa\rangle}_a m\langle\!\langle p : \kappa \rangle\!\rangle}$$

AAct Rcv

$$\frac{j \in I \quad \kappa_p \models \pi_j}{a[\Sigma_{i \in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i] \xrightarrow{m\langle p:\kappa_p\rangle}_a a[P_j\{^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

AAct If$_t$

$$\frac{}{a[\text{if } m{:}\kappa = m{:}\kappa' \text{ then } P \text{ else } Q\,] \xrightarrow{\tau}_a a[P]}$$

AAct If$_f$

$$\frac{m \neq n}{a[\text{if } m{:}\kappa = n{:}\kappa' \text{ then } P \text{ else } Q\,] \xrightarrow{\tau}_a a[Q]}$$

AAct Res

$$\frac{S \xrightarrow{\alpha}_a S' \quad n \notin n(\alpha)}{(\nu n)S \xrightarrow{\alpha}_a (\nu n)S'}$$

AAct Open

$$\frac{S \xrightarrow{\overline{m}\langle n\rangle}_a S'}{(\nu n)S \xrightarrow{\overline{m}(n)}_a S'}$$

AAct Par

$$\frac{S \xrightarrow{\alpha}_a S' \quad bn(\alpha) \cap fn(T) = \emptyset}{S \mid T \xrightarrow{\alpha}_a S' \mid T}$$

AAct Comm

$$\frac{S \xrightarrow{\overline{m}\langle p:\kappa\rangle}_a S' \quad T \xrightarrow{m\langle p:\kappa\rangle}_a T'}{S \mid T \xrightarrow{\tau}_a S' \mid T'}$$

AAct Close

$$\frac{S \xrightarrow{\overline{m}(n:\kappa)}_a S' \quad T \xrightarrow{m\langle n:\kappa\rangle}_a T'}{S \mid T \xrightarrow{\tau}_a (\nu n)(S' \mid T')}$$

**Lemma A.13.** *$S \equiv_a T$ and $S \xrightarrow{\alpha}_a S'$ implies that $T \xrightarrow{\alpha}_a T'$ for some $T'$ such that $S' \equiv_a T'$.*

**Lemma A.14.** *$S \xrightarrow{\tau}_a S'$ implies that $S \rightarrow_a S'$.*

**Lemma A.15.** *The labelled semantics satisfies the following properties:*

- *$S \xrightarrow{\overline{m}\langle v\rangle}_a S'$ implies that $S \equiv_a m\langle\!\langle v\rangle\!\rangle \mid S'$.*
- *$S \xrightarrow{\overline{m}(v)}_a S'$ implies that $S \equiv_a (\nu n)(m\langle\!\langle v\rangle\!\rangle \mid S')$.*

**Lemma A.16.** *$S \rightarrow_a S'$ implies that $S \xrightarrow{\tau}_a S''$ for some $S''$ such that $S' \equiv_a S''$.*

**Corollary A.17.** *$S \rightarrow_a S'$ if and only if $S \xrightarrow{\tau}_a S''$ and $S'' \equiv_a S'$.*

*Proof.* Follows from Lemma A.14 and Lemma A.16. □

*Proof.* The proof for each direction of the double implication proceeds by rule induction. The details of the proof are standard and hence are omitted. □

**Definition A.18** (Asynchronous bisimilarity)**.** (Annotated, weak) asynchronous bisimilarity, $\approx_a$, is the largest symmetric binary relation on systems such that whenever $S \approx_a T$, it holds that if $S \xrightarrow{\alpha}_a S'$ then $T \xrightarrow{\alpha}_a T'$ for some $T'$ such that $S' \approx_a T'$.

**Lemma A.19.** *$S \equiv_a T$ implies $S \approx_a T$.*

**Theorem A.20.** *$S \approx_a T$ implies $S \cong_a T$.*

*Proof.* We need to show that $\approx_a$ satisfies the three defining criteria of $\cong_a$, that is, preservation of barbs, closure under reduction and contextuality. From that, and the fact that $\cong_a$ is by definition the largest relation satisfying those three conditions, it will follow that $\approx_a \subseteq \cong_a$, i.e., $S \approx_a T$ implies $S \cong_a T$. □

**Theorem A.21.** *$S \cong_a T$ implies $S \approx_a T$.*

**Corollary A.22.** *$\cong_a$ and $\approx_a$ coincide.*

*Proof.* The proof follows directly from Theorem A.20 and Theorem A.21. □

# Appendix B

# The Provenance Calculus with Filters

Chapter 7 extended the provenance calculus with filters. It only described how this extension differs from the original calculus. The aim of this appendix is to provide a reference containing the full definitions of the syntax and semantics of the provenance calculus extended with filters.

## B.1 Syntax

The syntax of the calculus differs from that of the annotated version in the output construct (which now takes a filter as a second argument) and in provenance sequences (which now include a new form to denote the application of a filter to a provenance sequence). The full syntax is summarised in Figure B.1.

Figure B.1. *Syntax of the provenance calculus with filters*

| | |
|---|---|
| $\phi, \psi, \ldots \in \Phi$ | filters |
| $P ::=$ | process terms |
| $w\,\langle w\,\mathsf{at}\,\phi\rangle$ | Filtered output |
| $\Sigma_{i\in I} w\,(\pi_i\ \mathsf{as}\ x).P_i$ | input-guarded choice |
| $\mathsf{if}\ w = w'\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | equality test of $w$ and $w'$ |
| $(\nu n)P$ | private channel $n$ with scope $P$ |
| $P \mid Q$ | parallel composition of $P$ and $Q$ |
| $* P$ | replication of $P$ |

| | |
|---|---|
| $\kappa ::=$ | provenance sequences |
| $\epsilon$ | empty provenance |
| $e$ | single event |
| $\kappa \,;\, \kappa$ | sequential composition |
| $\phi \triangleright \kappa$ | filtered sequence |

## B.2 Semantics

We give a quick review of the complete definitions of both the structural congruence and reduction relations here. These are given below in figures B.2 and B.3 respectively.

The structural congruence relation is exactly the same as the one for the original calculus. We name the rules differently to avoid confusion.

Figure B.2. *Filtered semantics: structural congruence*

| | |
|---|---|
| (FStr Par Nil) | $S \mid 0 \equiv_f S$ |
| (FStr Par Comm) | $S \mid T \equiv_f T \mid S$ |
| (FStr Par Assoc) | $(R \mid S) \mid T \equiv_f R \mid (S \mid T)$ |
| (FStr Res Nil) | $(\nu n)0 \equiv_f 0$ |
| (FStr Res Res) | $(\nu n)(\nu m)S \equiv_f (\nu m)(\nu n)S$ |
| (FStr Res Par) | $((\nu n)S) \mid T \equiv_f (\nu n)(S \mid T) \quad \text{if } n \notin fn(T)$ |
| (FStr Sys Rep) | $a[* P] \equiv_f a[P \mid * P]$ |
| (FStr Sys Nil) | $a[0] \equiv_f 0$ |
| (FStr Sys Par) | $a[P \mid Q] \equiv_f a[P] \mid a[Q]$ |
| (FStr Sys Res) | $a[(\nu n)P] \equiv_f (\nu n)(a[P])$ |
| (FStr Par SRef) | $S \equiv_f S$ |
| (FStr Par SSym) | $S \equiv_f T \Rightarrow T \equiv_f S$ |
| (FStr Par STran) | $R \equiv_f S \wedge S \equiv_f T \Rightarrow R \equiv_f T$ |
| (FStr Par SCon) | $S \equiv_f T \Rightarrow C[S] \equiv_f C[T]$ |

Below we give the complete definition of reduction for the filtered calculus. The two main rules, and the only two that differ from the original calculus are FRed Snd and FRed Rcv which we already described in the previous chapter. The other rules stay unchanged.

Figure B.3. *Filtered semantics: provenance tracking reduction*

FRed Snd

$$\frac{}{a[m{:}\kappa_m\,\langle p{:}\kappa_p \text{ at } \phi\rangle] \longrightarrow_f m\langle\!\langle p{:}\phi \triangleright a!\kappa_m\,;\,\kappa_p\rangle\!\rangle}$$

FRed Rcv

$$\frac{j \in I \qquad view(a, \kappa_p) \leq \pi_j}{a[\Sigma_{i\in I} m{:}\kappa_m\,(\pi_i \text{ as } x).P_i] \mid m\langle\!\langle p{:}\kappa_p\rangle\!\rangle \longrightarrow_f a[P_j\{{}^{p:a?\kappa_m;\kappa_p}/_x\}]}$$

FRed If$_t$

$$\frac{}{a[\text{if } m{:}\kappa = m{:}\kappa' \text{ then } P \text{ else } Q\,] \longrightarrow_f a[P]}$$

FRed If$_f$

$$\frac{m \neq n}{a[\text{if } m{:}\kappa = n{:}\kappa' \text{ then } P \text{ else } Q\,] \longrightarrow_f a[Q]}$$

FRed Res

$$\frac{S \longrightarrow_f S'}{(\nu n)S \longrightarrow_f (\nu n)S'}$$

FRed Par

$$\frac{S \longrightarrow_f S'}{S \mid T \longrightarrow_f S' \mid T}$$

FRed Str

$$\frac{S \equiv_a T \quad T \longrightarrow_f T' \quad T' \equiv_a S'}{S \longrightarrow_f S'}$$

# Bibliography

[1] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. In *Proceedings of the First international conference on Principles of Security and Trust*, POST'12, pages 410–429, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28640-7.

[2] Lucia Acciai and Michele Boreale. Spatial and behavioral types in the pi-calculus. *Inf. Comput.*, 208(10):1118–1153, October 2010. ISSN 0890-5401.

[3] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics (Studies in Logic and the Foundations of Mathematics, Volume 103). Revised Edition*. North Holland, revised edition, November 1985. ISBN 0444875085.

[4] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: databases with uncertainty and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 953–964. VLDB Endowment, 2006.

[5] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 900–911. VLDB Endowment, 2004. ISBN 0-12-088469-0.

[6] Gerard Boudol. Asynchrony and the $\pi$-calculus. Technical Report 1702, INRIA, Sophia-Antipolis, 1992.

[7] Uri Braun, Avraham Shinnar, and Margo Seltzer. Securing provenance. In *The 3rd USENIX Workshop on Hot Topics in Security*, USENIX HotSec, pages 1–5, Berkeley, CA, USA, July 2008. USENIX Association.

[8] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, volume 4353 of *LNCS*, pages 209–223. Springer, 2007. ISBN 3-540-69269-X.

[9] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001. ISBN 3-540-41456-8.

[10] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 150–158, New York, NY, USA, 2002. ACM. ISBN 1-58113-507-6.

[11] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995. ISSN 0304-3975.

[12] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. *Theoretical Computer Science*, 398(1-3):217–242, 2008.

[13] James Cheney. A formal framework for provenance security. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 281–293, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4365-9.

[14] James Cheney. Semantics of the PROV Data Model. Technical report, W3C, 2013.

[15] James Cheney, Amal Ahmed, and Umut Acar. Provenance as dependency analysis. *Database Programming Languages*, 4797:138–152, 2007.

[16] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, April 2009. ISSN 1931-7883.

[17] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: a future history. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 957–964, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4.

[18] James Cheney, Paolo Missier, and Luc Moreau. Constraints of the Provenance Data Model. Technical report, W3C, 2013.

[19] Yingwei Cui and Jennifer Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378. IEEE Computer Society, March 2000. ISBN 0-7695-0506-6.

[20] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, and Sarah Cohen Boulakia. Privacy issues in scientific workflow provenance. In *Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, Wands '10, pages 3:1–3:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0188-6.

[21] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, and Yi Chen. On provenance and privacy. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 3–10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0529-7.

[22] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Statistical and Scientific Database Management*, pages 37–46. IEEE Computer Society, 2002. ISBN 0-7695-1632-7.

[23] James Frew and Rajendra Bose. Earth system science workbench: A data management infrastructure for earth science products. In *Statistical and Scientific Database Management*, pages 180–189. IEEE Computer Society, 2001. ISBN 0-7695-1218-6.

[24] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1):32–66, 1997.

[25] Floris Geerts and Jan Van den Bussche. Relational completeness of query languages for annotated databases. In Marcelo Arenas and Michael I. Schwartzbach, editors, *Database Programming Languages, 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers*, volume 4797 of *Lecture Notes in Computer Science*, pages 127–137. Springer, 2007. ISBN 978-3-540-75986-7.

[26] Yolanda Gil, James CVheney, Paul Groth, Olaf Hartig, Simon Miles, Luc Moreau, and Paulo Pinheiro da Silva. Provenance xg final report. Technical report, World Wide Web Consortium, 2010.

[27] Yolanda Gil, Simon Miles, Khalid Belhajjame, Helena Deus, Daniel Garijo, Graham Klyne, Paolo Missier, Stian Soiland-Reyes, and Stephen Zednik. PROV Model Primer. Technical report, W3C, 2013.

[28] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007. ISBN 978-1-59593-685-1.

[29] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 675–686. VLDB Endowment, 2007. ISBN 978-1-59593-649-3.

[30] Mark Greenwood, Carole Goble, Robert Stevens, Jun Zhao, Matthew Addis, Darren Marvin, Luc Moreau, and Tom Oinn. Provenance of e-science experiments - experience from bioinformatics. Proceedings of the UK OST e-Science Second All Hands Meeting, January 2003.

[31] Paul Groth, Steve Munroe, Simon Miles, and Luc Moreau. *In Lucio Grandinetti (ed.), HPC and Grids in Action*, chapter Applying the Provenance Data Model to a Bioinformatics Case. IOS Press, January 2008.

[32] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *Proceedings of the 2007 ACM workshop on Storage security and survivability*, StorageSS '07, pages 13–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-891-6.

[33] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *Trans. Storage*, 5(4):12:1–12:43, December 2009. ISSN 1553-3077.

[34] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP*, pages 133–147. Springer-Verlag, 1991. ISBN 3-540-54262-0.

[35] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching. In *POPL*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[36] Hook Hua, Curt Tilmes, Stephan Zednik, and Luc Moreau. PROV-XML: The PROV XML Schema. Technical report, W3C, 2013.

[37] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004. ISBN 052154310X.

[38] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[39] T. Imieliński and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, October 1984.

[40] Graham Klyne, Paul Groth, Luc Moreau, Olaf Hartig, Yogesh Simmhan, James Myers, Timothy Lebo, Khalid Belhajjame, and Simon Miles. PROV-AQ: Provenance Access and Query. Technical report, W3C, 2013.

[41] Marta Kwiatkowska and Vladimiro Sassone. GC2: Science for global ubiquitous computing. *Grand Challenges in Computing*, 2005.

[42] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. ProbView: A flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, September 1997.

[43] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. PROV-O: The PROV Ontology. Technical report, W3C, 2013.

[44] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-71499-X.

[45] Cédric Lhoussaine and Vladimiro Sassone. A dependently typed ambient calculus. In *ESOP*, volume 2986 of *LNCS*, pages 171–187. Springer, 2004.

[46] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[47] Robin Milner. Ubiquitous computing: Shall we understand it? *The Computer Journal*, 49(4):383–389, July 2006. ISSN 0010-4620.

[48] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401.

[49] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992. ISSN 0890-5401.

[50] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 685–695, London, UK, 1992. Springer-Verlag. ISBN 3-540-55719-9.

[51] Luc Moreau. The foundations for provenance on the web. *Found. Trends Web Sci.*, 2(2&#8211;3):99–241, February 2010. ISSN 1555-077X.

[52] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance

model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743 – 756, 2011. ISSN 0167-739X.

[53] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The Provenance of Electronic Data. *Communications of the ACM*, April 2008.

[54] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. Technical report, W3C, 2013.

[55] Luc Moreau, Paolo Missier, James Cheney, and Stian Soiland-Reyes. PROV-N: The Provenance Notation. Technical report, W3C, 2013.

[56] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX, 2006.

[57] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846286913.

[58] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima Press, 2011. ISBN 9780981531649.

[59] OED Online. Provenance, May 2009.

[60] C Pancerella, Jim Myers, and L Rahn. Data provenance in the collaboratory for multiscale chemical science (CMCS). *Workshop on Data Derivation and Provenance*, Jan 2002.

[61] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[62] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

[63] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2 edition, September 2008. ISBN 0596510330.

[64] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[65] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. ISBN 9781107003637.

[66] Davide Sangiorgi and David Walker. *The $\pi$-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[67] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005. ISSN 0163-5808.

[68] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. myGrid: personalised bioinformatics on the information grid. In *ISMB (Supplement of Bioinformatics)*, pages 302–304, 2003.

[69] Y. Richard Wang and Stuart E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538. Morgan Kaufmann Publishers Inc., 1990. ISBN 1-55860-149-X.

[70] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3): 94–104, 1991.

[71] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of the Thirteenth International Conference on Data Engineering*, pages 91–102. IEEE Computer Society, 1997. ISBN 0-8186-7807-0.