Università degli Studi di Pisa

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

Tesi di laurea magistrale

Security Issues in Adaptive Programming

Candidato:

Francesco Salvatori

Matricola 452095

Relatori:

Prof.ssa Chiara Bodei

Prof. Pierpaolo Degano

Dott. Letterio Galletta

Anno Accademico 2012–2013

Abstract

Adaptive systems improve their efficiency by modifying their behaviour to respond to changes of their operational environment. Also security must adapt to these changes and enforcing policies becomes dependent on the dynamic contexts. We address some issues of context-aware security from a language-based perspective. More precisely we extend a core adaptive functional language, recently introduced, with primitives to enforce security policies on the code execution. We then accordingly extend the existing static analysis in order to insert checks in a program so to guarantee that no violation occurs of the required security policies.

Contents

Chapter 1

Introduction

1.1 Context and Adaptivity

Today's software systems are expected to operate *every time* and *everywhere*: they have therefore to cope with changing environments, without compromising the correct behaviour of applications and without breaking the guarantees on their non-functional requirements, e.g., security or quality of service. As a consequence, software needs effective mechanisms to sense the changes of the operational environment, namely the *context*, in which the application is plugged in, and to properly *adapt* to changes. At the same time, these mechanisms must maintain the functional and non-functional properties of applications after the adaptation steps.

The context is a key notion for adaptive software. It is usually a complex entity, independent from the single applications, that includes different kinds of information coming both from outside (e.g., sensor values, available devices, code libraries etc. offered by the environment), and from inside the application boundaries (e.g., its private resources, user profiles, etc.).

Context Oriented Programming (COP), introduced in [13], is a recent paradigm that explicitly deals with contexts and provides programming adaptation mechanisms to support dynamic changes of behaviour, in reaction to changes in the context. Also subsequent work [23, 2, 27, 3] follow this approach to address the design and the implementation of concrete programming languages. The notion of context-dependent *behavioural variation* is central to this paradigm: it is a chunk of behaviour that can be activated depending on the current context hosting the application, so to dynamically modify the execution.

1.2 Security and Contexts

Security is one of the challenges arising in context-aware systems. The combination of security and context-awareness requires to address two distinct and interrelated aspects. On the one side, security requirements may reduce the

adaptivity of software, by adding further constraints on its possible actions. On the other side, new highly dynamic security mechanisms are needed to scale up to adaptive software. Such a duality has already been put forward in the literature [47, 10], that presents two ways of addressing it: *securing context-aware systems* and *context-aware security*.

Securing context-aware systems aims at rephrasing the standard notions of confidentiality, integrity and availability [40] and at developing techniques for guaranteeing them [47]. The challenge is to understand how to get secure and trusted context information. Contexts may contain indeed sensible data of the working environment (e.g., information about surrounding digital entities) that should be protected from unauthorised access and modification, in order to grant confidentiality and integrity. A trust model is therefore needed.

Context-aware security is dually concerned with the use of context information to drive security decisions. It has therefore to do with the definition and enforcement of high-level policies that talk about, are based and depend on the notion of dynamic context. Consider, for instance, the usual no flash photography policy in museums. A standard security policy does *not* allow people to take pictures, using the flash. A context-aware security is more flexible: it instead forbids flashing *only* inside those rooms that exhibit delicate paintings.

Most of the related work aims at implementing mechanisms at different levels of the infrastructures, e.g., in the middleware [42] or in the interaction protocols [22]. More foundational issues have instead been studied less. Moreover, the two dual aspects of context-aware security sketched above are often tackled separately, thus we still lack a unifying concept of security. Also, in the adaptive framework the most relevant concern is access control, see e.g. [47, 25, 49].

1.3 Our proposal

The kernel of our proposal is $\mathrm{ML}_{\mathrm{CoDa}}$, a core of ML extended with COP features. Its main novelty is to be a two-component language: a declarative constituent for programming the context and a functional one for computing (see [20] for details about its design).

The context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented as a Datalog program [39, 33]. Applications can therefore query the context by simply verifying whether a given property holds in it, in spite of the fact that this may involve possibly complex deductions.

Programming adaptation is specified through behavioural variations, that are activated depending on information picked up from the context, so to dynamically modify the execution. Differently from other proposals, in $\mathrm{ML}_{\mathrm{CoDa}}$ behavioural variations are first class, higher-order constructs: they can be referred to by identifiers, and passed as argument to, and returned by functions. This makes it possible to program dynamic and compositional adaptation patterns, as well as reusable and modular code.

As a matter of fact $\mathrm{ML}_{\mathrm{CoDa}}$, as it is, offers the features needed for addressing context-aware security issues, in particular for defining access control policies

and for enforcing them. First, we can express *system-defined* policies in *stratified Datalog with negation*, which is one of the two components of $\mathrm{ML_{CoDa}}$. This version of Datalog is sufficiently expressive for our policies. It is powerful enough to express all relational algebra [18] and in addition it is fully decidable and guarantees polynomial response time. Furthermore, adopting a stratified-negation-model is common and many logical languages for defining control access policies compile in Stratified Datalog, e.g., [9, 32, 17].

Secondly, the *dispatching* mechanism of $\mathrm{ML_{CoDa}}$, originally designed for selecting the right action to perform in behavioural variations, suffices for checking whether a specific policy holds. Therefore, our language requires no extensions to deal with security policies.

Actually, we can distinguish two classes of policies: those specified by the system to control the user's behaviour, and those expressed by the application. We are only interested in system policies. This is because the application developer has indeed full knowledge of his policies, and so he can specify them as behavioural variation constructs. Instead, the application has no *a priori* knowledge about the policies that contexts may require; then the system has no warranty that the application was designed to comply with them.

In the world of "secure" adaptive software, a runtime error can occur because of two different reasons, besides the presence of bugs in the code. An application can fail because it cannot adapt to the current context (*functional failure*) or because it violates a policy (*non-functional failure*). We would like to predict as earlier as possible if either case may occur. Note that some information about the running context is only available when the application is linked with it, so a fully static approach is not possible. For this reason we have a two-phase verification: one at compile time and one at linking time.

This is the approach followed in [19], that proposes a two-phase static technique for verifying whether a program adequately reacts to all context changes, signalling possible functional failures. The first phase is based on a type and effect system, that safely computes an approximation of the behaviour of the application at compile time. This approximation is then used at linking time to verify that the resources needed by the application to run will always be available in the actual context, and in its future modifications.

We extend here this technique to prevent an application from violating the required policies. A fully static yes/no procedure may lead to reject too many applications, so our extension is designed to also provide us with the means for instrumenting the code with suitable checks aimed at guarding the activities that can be considered risky. Actually, we have a sort of *runtime monitor* that is switched on and off at need, and, again, the dispatching mechanism of $\mathrm{ML_{CoDa}}$ suffices for natively supporting it.

After introducing some background about adaptive programming and static analisys techniques (Chapter 2), we will introduce $\mathrm{ML_{CoDa}}$ and our proposal with the help of a running example (Chapter 3), along with an intuitive presentation of the various components of our compile time and linking time static analysis, as well as of how security is dynamically enforced. The formal defi-

nitions and the statements of the correctness of our proposal will follow in the remaining chapters. The conclusion summarises our results and discusses some future work.

Chapter 2

Preliminary work

2.1 Adaptive Software

In this section we introduce the notion of adaptive software (see [43] for a complete survey). There are many definition of self-adaptability. For example, the one given in [38]:

> Self-adaptive software modifies its own behaviour in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.

The problem of self-adaptability been tackled by a variety of points of view such as control theory [30], artificial intelligence [28], software engineering [1, 41] and programming languages. We follow this last line of research. Language based approaches push the adaptation down to elementary software components: at this level we can describe extremely fine-grain adaptability mechanisms and introduce verification techniques, as those we will define in the next chapters. Among these approaches, we adopt *Context Oriented Programming* [24] because it has an explicit notion of context, which better models the working environment hosting the software.

2.1.1 Context-Oriented Programming

Context-Oriented Programming [14, 24] is a paradigm recently proposed as a viable approach to development of systems that are context-aware, i.e. able to dynamically adapt their behaviour depending on changes occurring in their execution environment.

Context-oriented languages are designed with suitable constructs for adaptation that express context-depended behaviour in a modular and isolating manner. This enables optimizations of code by the compiler or by the vir-

tual machine and also automatised verification to assure that programs keep their correctness after the adaptation.

Most of the existing languages that follows the COP paradigm are based on two fundamental concepts: *behavioural variations* and *layers*. A behavioural variation is a chunk of behaviour that can be *dynamically activated* depending on information picked up from the context, so to modify the application behaviour at runtime. Furthermore, multiple behavioural variations can be active at the same time, and the result of their *combination* determines the actual program behaviour.

A layer is an elementary property of the context. Indeed, the context is a set of layers, that can be activated and deactivated at runtime. Layers have to be first class objects[1] in the language, i.e. they can be bound to variables, passed as argument and retuned by functions. This feature is especially required to allow different parts of a system to communicate and perform runtime adaptability.

Inspired by the pioneering work of [24], a large number of COP proposals emerged, each of which addresses the problem of behavioural variation modularization and dynamic layer activation in different ways. See [44] for more details about the different language proposals, [4] for an analysis of some implementations and their performance and [21] for an comparison about the implementation of adaptive software by exploiting traditional languages and COP languages.

2.1.2 Open issues in context-oriented languages

So far most of the research efforts in the field of Context-Oriented Programming has been directed toward the design and the implementation of concrete languages. To the best of our knowledge only a limited number of papers in the literature have a precise semantic description of these languages.

We identify the following restrictions as the most limiting for the development of complex adaptive software:

1. Neither semantics foundation nor verification mechanism for this class of languages have been extensively studied so far.

2. Primitives used to describe the context are too low-level. Therefore, they are sometimes inadequate to program complex adaptive applications, where the context may contain structured information.

3. Behavioural variations are not first class citizens in the languages and one cannot easily manipulate them, e.g. passing them to functions as arguments, or binding them to variables.

4. Finally, security issues have been scarcely considered within COP languages.

[1] Note that in Chapter 4 we will introduce also behavioural variations as first class objects, enriching the number of adaptation patterns we can express.

In the following chapters we describe $\mathrm{ML}_{\mathrm{CoDa}}$ [20], a different approach that mainly deals with (1), (2) and (3) by introducing a completely new treatment of COP primitives (see Chapters 4, 5 and 6). Then, we try to address the weakness pointed out in (4) by properly extending this language: security issues will be considered in Chapters 7 and 8.

2.2 Static analysis techniques

In this section, we briefly review the concepts and methods underlying the static analysis techniques that we intend to exploit.

The purpose of static analysis is to acquire information about the runtime behaviour of a program without actually executing it but only by examining its source code. The results of an analysis can be used to optimize the execution, to discover errors in the code or to mathematically prove properties about programs. Typical examples are data-flow analysis [35, 29], Flow Logic [37, 8], type systems and their extensions (type and effect systems [36], dependent types [48], refinement types [7]), model-checking [12, 26], abstract interpretation [15, 16].

In the rest of this section we will explain in more detail the ideas underlying type systems and Flow Logic to make the formal development in later chapters clearer.

2.2.1 Type Systems

Among formal methods used to ensure that a system behaves correctly, type systems are among the most popular and the most largely used. They belong to the class of deductive systems where the proved theorems are about the types of programs. Type systems are made up of the elements below.

Type syntax and type environment The type syntax describes the basic types and the type constructors which can be used to obtain new types from existent ones. A type environment stores the associations between variables and the types of the values which may denote. Formally, it is a list of pair recursively defined as $\Gamma ::= \emptyset \mid \Gamma, \mathtt{x} : \mathtt{t}$.

Judgements Judgements are formal assertions about the typing of program phrases and correspond to the well-formed formulas of the deduction systems. A typical judgement has the form $\Gamma \vdash \mathcal{A}$, meaning that Γ entails $\mathcal{A}$, where $\mathcal{A}$ is an assertion whose free variables are bound in Γ. Usually, $\mathcal{A}$ asserts that there exists a relationship between a program phrase (*has-type judgement*) or between two types (*subtype-of judgement*).

Typing rules Typing rules establish relationships among judgements and assert the validity of a certain judgement, the *conclusion*, on the basis of others judgements that are already known to be valid, the *premises*. Also, we have

rules without premises, called *axioms*. They assert judgement that are always valid. We can repetely apply typing rules and compose them into derivations.

Soundness theorem A formal proof of a soundness theorem is required to guarantee that the type system achieves his objectives, that is, no type error occurs at runtime. If the semantics is specified with an operational style, usually the soundness theorem is proved in two steps: (*i*) *progress*, that is, a well-typed phrase is a value or it can take a step according to the evaluation rules and (*ii*) *preservation*, that is, if a well-type phrase takes a step of evaluation the resulting phrase is also well-typed (types are preserved under reduction).

2.2.2 Type and Effect Systems

Type and effect systems are a powerful extension of type systems which allows one to express general semantic properties and to statically reason about program execution. They compute the type of each program sentence and an approximate (but sound) description of its runtime behaviour.

The elements of a type system are extended as follows:

- in type syntax we annotate types with effects or tags describing some semantic properties. For example, $\mathtt{t}_1 \xrightarrow{\varphi} \mathtt{t}_2$ means that the functional types are annotated with the effect φ that will hold after the function application;
- judgements also express properties about runtime behaviour. For example, $\Gamma \vdash \mathtt{M} : \mathtt{t} \triangleright \varphi$ means that in the environment Γ, $\mathtt{M}$ has type $\mathtt{t}$ and that the effect described by φ holds.
- the correctness of a type and effect system is proved in two steps: (*i*) prove that the type and effects system is a conservative extension of the underlying type system; (*ii*) prove a soundness theorem that consider the effects.

2.2.3 Flow Logic

Flow Logic [37] is a declarative approach based on logical systems. Its distinctive feature is to separate the *specification* of the analysis from its actual *computation*. Intuitively, the specification describes when the results, namely analysis *estimates*, are *acceptable*, i.e. describing the property which we are concerned with. Furthermore, Flow Logic provides us with a methodology to define a correct analysis algorithm which operates in polynomial time, by reducing the specification to a constraints satisfaction problem.

Adopting a syntax-directed style (see [37] for a description of all possible styles), the analysis will be inductive defined by a set of inference rules. The Flow Logic methodology requires to prove that the analysis enjoys the two following properties: (*i*) the analysis is correct with respect to the dynamic semantics and (*ii*) every expression has a best or most precise analysis estimate.

The statement of the correctness theorem and its prove depend on the style used to specify the semantics. Adopting small step operational semantics, the correctness result is a subject reduction one, expressing that the analysis estimate remains acceptable under reduction. For (ii) is sufficient to prove that the set of the acceptable analysis estimates is a Moore family. s

2.2.4 Language-based Security

Traditionally, *security* has been considered an external property of programs. With the ubiquity of the Internet and of mobile computing devices it turns out that security is a fundamental concern, and that has to be taken into account from the first steps of the development process.

Language-based security [31, 45] has been proposed to address security concerns within programming languages. Nowadays it is a wide research area, that can be summarized by the following two points: (i) the usage of techniques from compilers, from static analysis and from program transformations to enforce and verify security properties, and (ii) the introduction into languages of constructs for dealing with security issues.

Another interesting approach is the one proposed in [6, 5, 46]. They introduced the notion of program history and the programming construct called security framing. A history is the sequence of actions that the program carries out at runtime. The security framing is a construct that allows programmers to enforce a security policy ϕ on a program fragment e (in symbols $\phi[e]$). Intuitively, it works as a monitor: after each execution step the framing requires that the current history η satisfies ϕ (written $\eta \models \phi$).

Chapter 3

Running example

We illustrate our methodology by considering a multimedia guide to a museum implemented as a smartphone application, starting from the case study of [20]. Assume the museum has a wireless infrastructure exploiting different technologies, like WiFi, Bluetooth, Irda or RFID. When a smartphone is connected, the visitor can access the museum Intranet and its website, from which he can download information about the exhibit and further multimedia contents.

Each exhibit is equipped with a wireless adapter (Bluetooth, Irda, RFID) and a QR code, used to provide the guide with the URL of the exhibit. The URL is retrieved by using one of the above technologies, depending on the smartphone capabilities. If a Bluetooth adapter is available, the smartphone can directly download the URL; otherwise if the smartphone has a camera and a QR decoder, the guide can retrieve the URL by taking a picture of the code and by decoding it.

The context In $\mathrm{ML}_{\mathrm{CoDa}}$ the smartphone capabilities are stored in the context as Datalog clauses. Consider, e.g., the following clauses defining when the smartphone can either directly download the URL (the predicate `device(d)` holds whether the device `d` is available) or it can take the URL by decoding a picture (the parameter `x` in the predicate `use_qrcore` is a handle for using the decoder):

```
direct_comm() ← device(irda).
direct_comm() ← device(bluetooth).
direct_comm() ← device(rfid_reader).

use_qrcode(x) ← user_prefer(qr_code),
                qr_decoder(x),
                device(camera).
use_qrcode(x) ← qr_decoder(x),
                device(camera),
                ¬ device(irda),
                ¬ device(rfid_reader),
                ¬ device(bluetooth).
```

Adaptivity Contextual data, such as the above predicates `direct_comm`() and `use_qrcode`(`decoder`), affect the download. The program flow has to change in accordance to the current context. To achieve this we exploit behavioural variations, offered by the functional part of ML_{CoDa}. Syntactically they are similar to pattern matching, where Datalog goals replace patterns and where parameters can additionally occur. Just like functional abstractions, behavioural variations have to be applied. Their application triggers a *dispatching mechanism* that at runtime inspects the context and selects the first expression whose goal holds.

For example, in the following function `getExhibitData`, we declare a behavioural variation (called `url`) with no arguments. It returns the URL of an exhibit, retrieved depending on the smartphone capabilities. If the smartphone can directly download the URL, then it does, through the channel returned by the function `getChannel`(), otherwise the smartphone takes a picture of the QR code and decodes it. Note that, in this second case, the variables `decoder` and `cam` will be assigned the handles of the decoder and the one of the camera deduced by the Datalog machinery. These handles are used by the functions `take_picture` and `decode_qr` to interact with the actual smartphone resources.

```
fun getExhibitData () =
  let url = (){
    ← direct_comm().
        let c = getChannel () in
          receiveData c,
    ← use_qrcode(decoder), camera(cam).
        let p = take_picture cam in
          decode_qr decoder p }
  in
      getRemoteData #url
```

The behavioural variation (bound to) `url` is applied before invoking the function `getRemoteData`, that connects to the corresponding website and downloads the required information. Here, the application of a behavioural variation is represented by #; for details see Chapter 4.

Formally, suppose the current context C satisfies the goal $\leftarrow$ `direct_comm`(), and apply the function `getExhibitData` to *unit*. The computation acts as follows. Here, a transition $C,\ e \rightarrow C',\ e'$ means that the expression e is evaluated in the context C and reduces to e' changing the context C to C':

- C, `getExhibitData` $() \rightarrow C$, `getRemoteData` $\#u$, where u is the behavioural variation bound to `url` in the body of the function `getExhibitData`. This is a standard evaluation step for the `let` construct.
- C satisfies the goal $\leftarrow$ `direct_comm`(), so the dispatching mechanism selects the first expression of the behavioural variation u. Said n the value returned by `getChannel`, we have that C, `getRemoteData` $\#u \rightarrow$ C, `getRemoteData`(`receiveData` n).

Context update and security policies To update the context at runtime, ML_{CoDa} provides us with the constructs `tell` and `retract`, that add and remove

Datalog facts, respectively. Suppose, for instance, that the context stores information about the room in which the user is through the predicate `current_room`. If the user moves from the room *delicate paintings* to the one *sculptures*, the application updates the context by executing

```
retract current_room(delicate_paintings)
tell current_room(sculptures)
```

Assume now that one can take pictures in every room, but that in the rooms with delicate paintings it is forbidden to use the camera flash, not to damage the exhibits. This policy is specified by the museum (the system) and it must be enforced during the user's tour. Since policies predicate on the context, they are easily expressed as Datalog goals. Let the fact `flash_on` hold when the flash is active and the fact `button_clicked` when the user presses the button of the camera. The above policy Φ is then expressed in Datalog as the goal

```
phi ← ¬ current_room(delicate_paintings)
phi ← ¬ button_clicked
phi ← ¬ flash_on
```

that, intuitively, is the result of compiling the following logical condition:

$$current\_room(delicate\_paintings) \Rightarrow (button\_clicked \Rightarrow \neg flash\_on)$$

Of course, the museum can specify many other policies. We assume that there is a unique global policy Φ (referred to in the code as `phi`), obtained by suitably combining all the required policies. The enforcement is obtained by a runtime monitor that checks the validity of Φ every time the context changes, i.e., every time a `tell`/`retract` is performed. We remark that the introduction of the runtime monitor requires no modification of the language, because our policies are Datalog goals and can be checked by simply invoking the dispatching mechanism.

The need for a static analysis An application fails to adapt to a context (functional failure) when the dispatching mechanism fails, i.e., a behavioural variation gets stuck. Consider to evaluate `getExhibitData` on a smartphone without wireless technology and QR decoder. Clearly, no context will ever satisfy the goals of the behavioural variation `url`, thus, when `url` is applied no case can be selected.

Another kind of failure happens when an application violates a policy (non-functional failure). In our example, this happens when attempting to use the flash, if the context includes `current_room`(`delicate_paintings`).

To avoid functional failure and to optimise the policy enforcement, we equip $\mathrm{ML}_{\mathrm{CoDa}}$ with a two-phase static analysis: a type and effect system and a control-flow analysis. The analysis checks if an application will be able to adapt to its execution contexts and detects which actions can result in contexts that violate the required policies.

Type and effect system At *compile time* we associate to each expression e a type (which is almost standard) and an effect over-approximating its actual runtime behaviour. This effect is called *history expression*, and represents (an abstraction of) the interactions with the context that are performed during the evaluation of e.

To intuitively understand how this phase works, take the following expression, describing the actions for taking a picture:

```
e_a = let x =
        if always_flash then
          let y = tell F_1^1 in tell F_2^2
        else
          let y = tell F_1^3 in tell F_3^4
      in
        tell F_4^5
```

For clarity, here (and in the syntax in Chapter 4) we show the labels of `tell`/`retract` in the code. Actually, labels are inserted by the compiler during syntax analysis or type checking. Moreover, for the sake of readability the facts have been given a symbolic name. They are intended to be:

$$F_1 \equiv \texttt{photocamera\_started} \qquad F_2 \equiv \texttt{flash\_on}$$
$$F_3 \equiv \texttt{mode\_museum\_activated} \quad F_4 \equiv \texttt{button\_clicked}$$

The type of e_a is *unit* (that of `tell` $\texttt{F}_4$), and its history expression (also labelled) is

$$H_a = (((tell\ F_1^1 \cdot tell\ F_2^2)^3 + (tell\ F_1^4 \cdot tell\ F_3^5)^6)^7 \cdot tell\ F_4^8)^9$$

In history expressions $\cdot$ means sequential composition, while $+$ is for conditional expression.

Depending on the value of `always_flash`, which records whether the user wants the flash to be always usable, the expression e_a can either perform the action `tell` $\texttt{F}_1$ followed by `tell` $\texttt{F}_2$, *or* the action `tell` $\texttt{F}_1$ followed by `tell` $\texttt{F}_3$ — and the context is informed that the flash is on or off, respectively. After that, e_a will perform `tell` $\texttt{F}_4$, no matter what the previous choice was.

The labels of history expressions allow us to link the actions in histories to the corresponding points inside the code, e.g., the first `tell` $\texttt{F}_1$ in H_a, which is labelled 1, corresponds to the first `tell` $\texttt{F}_1$ in e_a, which is also labelled 1, while the `tell` $\texttt{F}_4$, labelled 8 in H_a, corresponds to that labelled 5 in e_a. More precisely, the correspondences are $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$; the abstract labels that do not annotate *tell/retract* constructs have instead no corresponding labels.

Control flow analysis The effects are exploited at *linking time* (i) to verify that the application can adapt to all contexts arising at runtime, and (ii) to identify which `tell`/`retract` are risky and need to be checked by the monitor. If our static analysis discovers that a `tell`/`retract` may lead to a violation, we have to activate the monitor during its evaluation, otherwise the monitor is

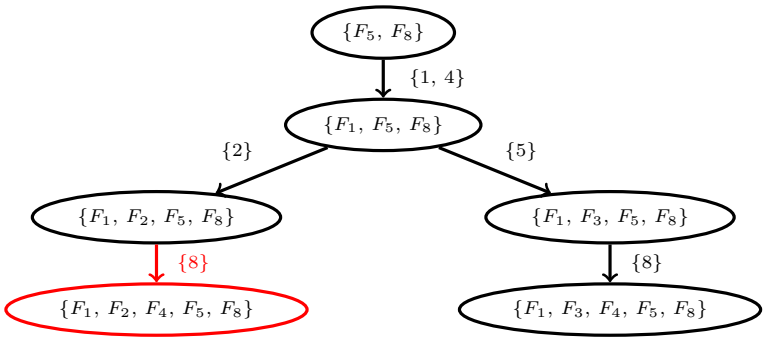

Figure 3.1: The evolution graph for the context $C \supseteq \{F_5,\ F_8\}$ and the history expression $H_a = (((tell\ F_1^1.tell\ F_2^2)^3 + (tell\ F_1^4.tell\ F_3^5)^6)^7.tell\ F_4^8)^9$

kept inactive. To do that, our control-flow analysis first builds a graph to trace how the initial context evolves during execution, and then it finds out in which contexts there is a violation and which operation might cause it.

Back to our example, consider an initial context C that includes the facts $F_8 \equiv$ `current_room(delicate_paintings)` and F_5 (irrelevant here), but not the facts $\{F_1, F_2, F_3, F_4\}$. Starting from C (and from the history expression H_a computed above) our loading time analysis builds the graph in Figure 3.1 (we show only the relevant facts of C). Nodes represent contexts, possibly reachable at runtime; edges represent transitions from one context to another. Each edge is annotated with the set of actions in H_a that may cause that transition. For instance, from the initial context it is possible to reach the context that also includes the fact F_1, because of the two *tell* operations labelled in the history expression by 1 and by 4. Therefore an edge can have more than one label (e.g., the one labelled $\{1, 4\}$). Note also that the same label may occur in more than one edge (e.g., the label 8).

As said, the labelling is done during the type checking and plays a key role in enforcing security policies. Here, we observe (e.g., by visiting the graph) that the context corresponding to the node $\{F_1,\ F_2,\ F_4,\ F_5,\ F_8\}$ (in red in Figure 3.1) violates our no-flash policy. This amounts to identifying a possible runtime violation. Since this node has a single incoming edge, labelled with 8 (highlighted in red in Figure 3.1, together with the corresponding abstract action), we can deduce that the possibly risky action is the corresponding dynamic `tell F4` labelled by 5 in the code. For preventing a violation, all we have to do is activating the runtime monitor right before executing this operation.

Chapter 4

$\mathrm{ML}_{\mathrm{CoDa}}$

The kernel of our proposal is $\mathrm{ML}_{\mathrm{CoDa}}$, a language consisting of two components: Datalog with stratified negation to describe both the context and the required security policies, and a core of ML extended with specific constructs which allows to express COP features. We present the syntax and the operational semantics of $\mathrm{ML}_{\mathrm{CoDa}}$; for more details and for a longer, fully worked out example consider that this language has been introduced for the very first time in [20] and its companion paper [19].

4.1 Syntax

$\mathrm{ML}_{\mathrm{CoDa}}$ consists of two sub-languages: a Datalog with negation to describe the context and a core ML extended with COP features.

The Datalog part is standard: a program is a set of facts and clauses. We assume that each program is safe [11]; to deal with negation, we adopt *Stratified Datalog* under the Closed World Assumption.

We enforce security properties by introducing policies Φ, expressed as Datalog goals, one of the components of $\mathrm{ML}_{\mathrm{CoDa}}$. As a consequence, the language requires no extensions to deal with security policies. The mechanism for selecting behavioural variations can indeed be also used for checking whether a specific policy holds, and for selecting the chunk of behaviour that does. We assume that the name `phi` is reserved for the overall security policy Φ expressed by the system. Thus, verifying whether the policy holds simply equals to check if the Datalog goal $\leftarrow$ `phi` is satisfied. As mentioned in Chapter 1, Stratified Datalog is sufficiently expressive for our policies. As a matter of fact, it is powerful enough to express all relational algebra [18] and in addition it is fully decidable and guarantees polynomial response time. Furthermore, many logical languages for defining control access policies compile in Stratified Datalog, e.g., [9, 32, 17].

The functional part inherits most of the ML constructs. In addition to the usual ones, our values include Datalog facts F and behavioural variations.

Moreover, we introduce the set $\tilde{x} \in DynVar$ of *parameters*, i.e., variables that assume values depending on the properties of the running context. We distinguish these parametres from the standard identifiers Var, thus requiring that $Var \cap DynVar = \emptyset$.
The syntax of ML_{CoDa} is below:

$$
\begin{aligned}
& x \in Var \quad \tilde{x} \in DynVar \quad (Var \cap DynVar = \emptyset) \\
Va ::= & G.e \mid G.e, Va \\
v ::= & c \mid \lambda_f x.e \mid (x)\{Va\} \mid F \\
e ::= & v \mid x \mid \tilde{x} \mid e_1\, e_2 \mid let\, x\, =\, e_1\, in\, e_2 \mid if\, e_1\, then\, e_2\, else\, e_3 \mid \\
& dlet\, \tilde{x}\, = e_1\, when\, G\, in\, e_2 \mid tell(e_1)^l \mid retract(e_1)^l \mid e_1 \cup e_2 \mid \#(e_1, e_2)
\end{aligned}
$$

To facilitate our static analysis (see Chapters 6 and 7) we require that each `tell`/`retract` in the code is uniquely and mechanically associated with a label $l \in Lab_C$. As usual, labels do not affect the dynamic semantics of the calculus.

Standard ML expressions need no explanation. Below, we comment the new COP- oriented constructs. First we have variations Va, i.e., lists of expressions $G_1.e_1, \ldots, G_n.e_n$ guarded by Datalog goals G_i, and behavioural variations $(x)\{Va\}$, each consisting of a variation Va. The variable x can freely occur in the expressions e_i. At runtime, the first goal G_i satisfied by the context determines the expression e_i to be selected (*dispatching*). Then we introduce context-dependent binding, that is the mechanism to declare variables whose values depend on the context. This is expressed with the *dlet* construct, which associates a parameter $\tilde{x}$ to a variation Va. It is syntactically similar to the standard let, but has additional Datalog goals that made the value of $\tilde{x}$ calculated depending on which goal is satisfied. We also have a way to update the context by asserting/retracting facts, provided that the resulting context satisfies the system policy Φ: the *tell*/*retract* constructs are provided for this purpose. Finally, we have operators for managing behavioural variations. The append operator $e_1 \cup e_2$ concatenates behavioural variations, so allowing for dynamic compositions. The application of a behavioural variation $\#(e_1, e_2)$, with $e_1 = (x)(Va)$, applies e_1 to its argument e_2. To do so, the dispatching mechanism is triggered to query the context and to select from Va the expression to run, if any. As in standard function application, the calculated value of e_2 is binded to the variable x, which may obviously occur in the body of Va.

4.2 Semantics

We now endow ML_{CoDa} with a small-step operational semantics.

For the Datalog evaluation we adopt the top-down standard semantics of stratified programs [11]. Given a context C and a goal G, $C \vDash G\, with\, \theta$ means that there exists a substitution θ, replacing constants for variables, such that the goal G is satisfied in the context C.

(If1)
$$\frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, if\, e_1\, then\, e_2\, else\, e_3 \to C',\, if\, e_1'\, then\, e_2\, else\, e_3}$$

(If2)
$$\frac{}{\rho \vdash C,\, if\, true\, then\, e_2\, else\, e_3 \to C',\, e_2}$$

(If3)
$$\frac{}{\rho \vdash C,\, if\, false\, then\, e_2\, else\, e_3 \to C',\, e_3}$$

(Let1)
$$\frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, let\, x\, =\, e_1\, in\, e_2 \to C',\, let\, x\, =\, e_1'\, in\, e_2}$$

(Let2)
$$\frac{}{\rho \vdash C,\, let\, x\, =\, v\, in\, e_2 \to C,\, e_2\{v/x\}}$$

(App1)
$$\frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, e_1\, e_2 \to C',\, e_1'\, e_2}$$

(App2)
$$\frac{\rho \vdash C,\, e_2 \to C',\, e_2'}{\rho \vdash C,\, (\lambda_f x.e)\, e_2 \to C',\, (\lambda_f x.e)\, e_2'}$$

(App3)
$$\frac{}{\rho \vdash C,\, (\lambda_f x.e)\, v \to C,\, e\{v/x, (\lambda_f x.e)/f\}}$$

Figure 4.1: The reduction rules for standard constructs of ML

The semantics of $\mathrm{ML}_{\mathrm{CoDa}}$ is defined for expressions with no free variable, but possibly with free parameters, allowing for openendness. For that, we have in an environment ρ, i.e., a function mapping parameters to variations $DynVar \to Va$.

A transition $\rho \vdash C,\, e \to C',\, e'$ says that in the environment ρ, the expression e is evaluated in the context C and reduces to e' changing the context C to C'. We assume that the initial configuration is $\rho_0 \vdash C,\, e_p$ where ρ_0 contains the bindings for all system parameters, and C results from the linking of the system context and of the application context.

Most of the rules of the small-step operational semantics are inherited from ML and presented in Figure 4.1. The inductive definitions of the rules for our new constructs are shown in Figure 4.2. Labels are not explicited, just because do not affect the semantics. We briefly comment below on the rules displayed.

Context-dependent binding The rules (Dlet1) and (Dlet2) for the construct *dlet*, and the rule (Par) for parameters implement our context-dependent binding. For brevity, we assume here that e_1 contains no parameters. The rule (Dlet1) extends the environment ρ by appending $G.e_1$ in front of the existent binding for $\tilde{x}$. Then, e_2 is evaluated under the updated environment. Notice that the *dlet* does not evaluate e_1 but only records it in the environment in a sort of call-by-name style. The rule (Dlet2) is standard: the whole *dlet* reduces to the value which eventually e_2 reduces to.

$$\text{(Par)}\quad \frac{\rho(\tilde{x}) = Va \qquad dsp(C,\, Va) = (e,\, \theta)}{\rho \vdash C,\, \tilde{x} \to C,\, e\,\theta}$$

$$\text{(Dlet1)}\quad \frac{\rho[(G.e_1, \rho(\tilde{x}))\,/\tilde{x}] \vdash C,\, e_2 \to C',\, e_2'}{\rho \vdash C,\, dlet\,\tilde{x} = e_1\, when\, G\, in\, e_2 \to C',\, dlet\,\tilde{x} = e_1\, when\, G\, in\, e_2'}$$

$$\text{(Dlet2)}\quad \frac{}{\rho \vdash C,\, dlet\,\tilde{x} = e_1\, when\, G\, in\, v \to C,\, v} \qquad \text{(Append1)}\quad \frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, e_1 \cup e_2 \to C',\, e_1' \cup e_2}$$

$$\text{(Append2)}\quad \frac{\rho \vdash C,\, e_2 \to C',\, e_2'}{\rho \vdash C,\, (x)\{Va_1\} \cup e_2 \to C',\, (x)\{Va_1\} \cup e_2'}$$

$$\text{(Append3)}\quad \frac{z \text{ fresh}}{\rho \vdash C,\, (x)\{Va_1\} \cup (y)\{Va_2\} \to C,\, (z)\{Va_1\{z/x\},\, Va_2\{z/y\}\}}$$

$$\text{(VaApp1)}\quad \frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, \#(e_1,\, e_2) \to C', \#(\,e_1',\, e_2)} \qquad \text{(VaApp2)}\quad \frac{\rho \vdash C,\, e_2 \to C',\, e_2'}{\rho \vdash C,\, \#((x)\{Va\},\, e_2) \to C', \#((x)\{Va\},\, e_2')}$$

$$\text{(VaApp3)}\quad \frac{dsp(C,\, Va) = (e,\, \{\overrightarrow{c}/\overrightarrow{y}\})}{\rho \vdash C,\, \#((x)\{Va\},\, v) \to C,\, e\{v/x,\ \overrightarrow{c}/\overrightarrow{y}\}}$$

$$\text{(Tell1)}\quad \frac{\rho \vdash C,\, e \to C',\, e'}{\rho \vdash C,\, tell(e) \to C',\, tell(e')} \qquad \text{(Tell2)}\quad \frac{dsp(C \cup \{F\},\, \leftarrow \texttt{phi}.()) = ((),\, \theta)}{\rho \vdash C,\, tell(F) \to C \cup \{F\},\, ()}$$

$$\text{(Retract1)}\quad \frac{\rho \vdash C,\, e \to C',\, e'}{\rho \vdash C,\, retract(e) \to C',\, retract(e')} \qquad \text{(Retract2)}\quad \frac{dsp(C \setminus \{F\},\, \leftarrow \texttt{phi}.()) = ((),\, \theta)}{\rho \vdash C,\, retract(F) \to C \setminus \{F\},\, ()}$$

Figure 4.2: The reduction rules for new constructs of $\mathrm{ML_{CoDa}}$

The (Par) rule looks for the variation Va bound to $\tilde{x}$ in ρ. Then the dispatching mechanism selects the expression to which $\tilde{x}$ reduces. The dispatching mechanism is implemented by the partial function dsp, defined as

$$dsp(C,\, (G.e, Va)) = \begin{cases} (e,\, \theta) & \text{if } C \vDash G\, with\, \theta \\ dsp(C,\, Va) & \text{otherwise} \end{cases}$$

It inspects a variation from left to right to find the first goal G satisfied by C, under a substitution θ. If this search succeeds, the dispatching returns the corresponding expression e and θ. Then $\tilde{x}$ reduces to $e\,\theta$, i.e., to e whose variables are bound by θ. Instead, if the dispatching fails because no goal holds, the computation gets stuck because the program cannot adapt to the current context. Our static analysis is also designed to prevent this kind of runtime errors.

As an example of context-dependent binding consider the simple conditional expression $\texttt{if}\,\tilde{x} = c_2\ \texttt{then}\ c_3\ \texttt{else}\ c_4$, in an environment ρ that binds the parameter $\tilde{x}$ to $e = G_1.\, c_1, G_2.\, c_2$ (c_i are constants) and in a context C that satisfies the goal G_2 but not G_1:

$$\begin{aligned} &\rho \vdash C,\, \texttt{if}\,\tilde{x} = c_2\ \texttt{then}\ c_3\ \texttt{else}\ c_4 \\ &\to C,\, \texttt{if}\, c_2 = c_2\ \texttt{then}\ c_3\ \texttt{else}\ c_4 \\ &\to C,\, c_3 \end{aligned}$$

In the first step, we retrieve the binding for $\tilde{\texttt{x}}$ (recall it is e), where $dsp(C,\, e) = dsp(C,\ (G_1.c_1, G_2.\, c_2)) = (c_2, \theta)$, for a suitable substitution θ.

Concatenating behavioural variations The rules for the concatenation $e_1 \cup e_2$ of two behavioural variations sequentially evaluate e_1 (rule (Append1)) and e_2 (rule (Append2)). Once they have been reduced to behavioural variations, they are concatenated together by renaming bound variables to avoid name captures (rule (Append3)).

As an example of behavioural variation concatenation, let $\texttt{T}$ be the goal always true, and consider the function $d = \lambda x.\lambda y.\, x \cup (w)\{\texttt{T}.y\}$. It takes as arguments a behavioural variation x and a value y, and extends x by adding a default case which is always selected when no other case apply. In the following computation we apply d to $p = (x)\{G_1.c_1, G_2.x\}$ and to c_2 (c_1, c_2 constants):

$$\begin{aligned} &\rho \vdash C,\, d\, p\, c_2 \\ &\to C,\, (x)\{G_1.c_1, G_2.x\} \cup (w)\{T.c_2\} \\ &\to C,\, (z)\{G_1.c_1, G_2.z, T.c_2\} \end{aligned}$$

Application of behavioural variations The application of the behavioural variation $\#(e_1, e_2)$ evaluates the subexpressions until e_1 reduces to $(x)\{Va\}$ (rule (VaApp1)) and e_2 to a value v (rule (VaApp2)). Then the rule (VaApp3) invokes the dispatching mechanism to select the relevant expression e from

which the computation proceeds after v is substituted for x. Also in this case the computation gets stuck if the dispatching mechanism fails. As an example, consider the behavioural variation $(x)\{G_1.c_1, G_2.x\}$ and apply it to the constant c in a context C that satisfies the goal G_2 but not G_1. Since $dsp(C, (x)\{G_1.c_1, G_2.x\}) = (x, \theta)$ for some substitution θ, we get

$$\rho \vdash C,\ \#((x)\{G_1.c_1, G_2.x\}, c)\ \rightarrow\ C,\ c$$

Updating the context The rules (Tell1) and (Retract1) evaluate the expression e until it reduces to a fact F, which is a value of $\mathrm{ML_{CoDa}}$. Then, the new context C', obtained from C by adding (respectively, removing) F, is checked against the security policy Φ (rules (Tell2) and (Retract2)). Since Φ is a Datalog goal, we can easily reuse our dispatching machinery, implementing the check as a call to the function *dsp* where the first argument is C' and the second one is the trivial variation $\leftarrow \texttt{phi}.()$. If this call produces a result[1], then the evaluation yields the unit value () and the new context C'.

The following example shows the reduction of a *retract* construct violating a policy Φ. Let the context be $C = \{F_3, F_4, F_5\}$. Suppose that Φ imposes the logical condition $F_3 \Rightarrow F_4$, and apply the function $\texttt{f} = \lambda\texttt{x}.\,\texttt{if}\ e\ \texttt{then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4$ to unit, assuming that the evaluation of e reduces to `false` without changing the context:

$$\begin{aligned}
&\rho \vdash C,\ \texttt{retract}(\texttt{f}\,())\\
&\rightarrow C,\ \texttt{retract}((\lambda\texttt{x}.\,\texttt{if}\ e\ \texttt{then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4)())\\
&\rightarrow C,\ \texttt{retract}(\texttt{if}\ e\ \texttt{then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4)\\
&\rightarrow C,\ \texttt{retract}(\texttt{if false then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4)\\
&\rightarrow C,\ \texttt{retract}(\texttt{F}_4)\\
&\nrightarrow
\end{aligned}$$

Since the policy requires that the fact F_4 always holds when F_3 holds too, the attempt to remove it from the context violates Φ. Consequently, the evaluation gets stuck because $dsp(C \setminus \{F_4\}, \leftarrow \texttt{phi}.())$ fails.

Instead, if e reduces to `true`, there is no violation of the policy and the evaluation reduces to unit:

$$\begin{aligned}
&\rho \vdash C,\ \texttt{retract}(\texttt{f}\,())\\
&\rightarrow C,\ \texttt{retract}(\texttt{if true then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4)\\
&\rightarrow C,\ \texttt{retract}(\texttt{F}_5)\\
&\rightarrow C \setminus \{\texttt{F}_5\}, ()
\end{aligned}$$

[1]Note that the result returned by the dispatching mechanism, if any, must consist of the *unit* expression () (that is, the one related to the goal $\leftarrow \texttt{phi}$ in the variation $\leftarrow \texttt{phi}.()$) and of a substitution θ (for the free variables possibly occuring in the goal Φ).

Chapter 5

Type and Effect System

We now associate ML_{CoDa} expressions with a type, an abstraction called *history expression*, and a function called *labelling environment*. During the verification phase the virtual machine uses this history expression to ensure that the dispatching mechanism will always succeed at runtime. Then, the labelling environment is used to drive us in instrumenting the code with security checks. First, we briefly present history expressions and labelling environments and then the rules of our type and effect system.

We remark that our type and effect system is largely inherited from that introduced in [19]: here we just extend it with the possibility of handling labelling environments.

5.1 History expressions and labelling environments

History expressions a simple process algebra used to soundly abstract the execution histories that a program may generate [6]. Here, history expressions approximate the sequence of actions that a program may perform over the context at runtime, i.e., asserting/retracting facts and asking if a goal holds.

To support the following formal development, we assume that history expressions are uniquely labelled on a given set Lab_H. Labels allow us to go back from static actions in histories to the corresponding actions inside the code. The syntax of history expressions is described below:

$$H ::= \mathfrak{z} \mid \epsilon^l \mid h^l \mid (\mu h.H)^l \mid tell\, F^l \mid retract\, F^l \mid (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \Delta$$
$$\Delta ::= (ask\, G.H \otimes \Delta)^l \mid fail^l$$

The empty history expression abstracts programs which do not interact with the context. For technical reasons, we syntactically distinguish when the empty history expression comes from the syntax (ϵ^l) and when it is instead obtained by reduction in the semantics ($\mathfrak{z}$). The history expression $\mu h.H$ represents

$$\frac{}{C,\ (\mathfrak{z} \cdot H) \to C,\ H} \qquad \frac{}{C,\ \epsilon \to C,\ \mathfrak{z}} \qquad \frac{}{C,\ tell\, F \to C \cup \{F\},\ \mathfrak{z}}$$

$$\frac{}{C,\ retract\, F \to C \backslash \{F\},\ \mathfrak{z}} \qquad \frac{C, H_1 \to C',\ H_1'}{C, (H_1 + H_2) \to C',\ H_1'}$$

$$\frac{C, H_2 \to C',\ H_2'}{C, (H_1 + H_2) \to C',\ H_2'} \qquad \frac{C,\ H_1 \to C',\ H_1'}{C,\ (H_1 \cdot H_2) \to C',\ (H_1' \cdot H_2)}$$

$$\frac{}{C, (\mu h.H) \to C, H[(\mu h.H)/h]} \qquad \frac{C \vDash G}{C,\ (ask\, G.H \otimes \Delta) \to C,\ H}$$

$$\frac{C \nvDash G}{C,\ (ask\, G.H \otimes \Delta) \to C,\ \Delta}$$

Figure 5.1: Semantics of History Expressions

possibly recursive functions, where h is the recursion variable; the "atomic" history expressions $tell\, F$ and $retract\, F$ are for the analogous expressions of $\mathrm{ML}_{\mathrm{CoDa}}$; the non-deterministic sum $H_1 + H_2$ abstracts the conditional expression *if-then-else*; the concatenation $H_1 \cdot H_2$ is for sequences of actions, that arise, e.g., while evaluating applications; Δ mimics our dispatching mechanism, where Δ is an *abstract variation*, defined as a list of history expressions, each element H_i of which is guarded by an $ask\, G_i$.

For example, the history expression computed for the behavioural variation `url` in the function `getExhibitData` of Section 3 is $H_{url} = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$, where the goals are $G_1 = \leftarrow \texttt{direct\_comm}()$ and $G_2 = \leftarrow \texttt{use\_qrcode}(\texttt{decoder}), \texttt{camera}(\texttt{cam})$ and where H_1 is the effect of the expression guarded by G_1 and H_2 is the effect of the one guarded by G_2. Intuitively, H_{url} says that at least one between G_1 or G_2 must be satisfied by the context in order to successfully apply the behavioural variation `url`.

Given a context C, the behaviour of a history expression H is formalised by the transition system inductively defined in Figure 5.1. Configurations have the form $C, H \to C', H'$ meaning that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones presented in [6]: below we only comment on those dealing with the context. Just as in the case of ML expressions, labels do not affect the semantic: for this reason they are not explicited in the figure.

An action $tell\, F$ reduces to $\mathfrak{z}$ and yields a context C' where the fact F has just been added; similarly for $retract\, F$. Differently from what we do in the semantic rules, here we do not consider the possibility of a policy violation: history expressions approximate how the application would behave in absence of any type of check. The rules for Δ scan the abstract variation and look for

the first goal G satisfied in the current context; if this search succeeds, the whole history expression reduces to the history expression H guarded by G; otherwise the search continues on the rest of Δ. If no satisfiable goal exists, the stuck configuration $fail$ is reached, to indicate that the dispatching mechanism fails.

Labelling Environment To support the implementation of security checks, we want to associate each abstract action described in a history expression to the corresponding concrete action that can be performed by the application during its execution. We are only interested in those expressions that have as an effect the addition or removal of a fact, so modifying the context. For this purpose we introduce an environment that maps (labels of) history expressions to (labels of) expressions in the code.

We assume as given the function $h : Lab_H \to H$, that recovers a construct in a given history expression starting from its label l. We do not need the constructs returned by h to be labelled. For example, given the simple history expression $H = (tell\, F_1^1 + tell\, F_2^2)^3$ we have that $h(1) = tell\, F_1$, $h(2) = tell\, F_2$ and $h(3) = tell\, F_1 + tell\, F_2$. This function h is only used to distinguish the type of the construct associated to a label; also, h will help us in some steps of static analysis, as we will see in the following chapters.

Now, we can define a way of going back from a `tell`/`retract` in a history expression to the corresponding operations in the code, by exploiting their labels in the set Lab_C (see Section 4). Formally, we introduce the function below:

Definition 5.1 (Labelling environment)**.** A *labelling environment* is a (partial) function

$$\Lambda : Lab_H \to Lab_C,$$

defined only if $h(l) \in \{tell(F), retract(F)\}$.

Just like history expressions, labelling environments will be computed by our type and effect system. To give an intuition of how labelling environments work, recall the history expression

$$H_a = (((tell\, F_1^1 \cdot tell\, F_2^2)^3 + (tell\, F_1^4 \cdot tell\, F_3^5)^6)^7 \cdot tell\, F_4^8)^9$$

of Section 3, obtained from the expression

```
e_a = let x =
        if always_flash then
          let y = tell F_1^1 in tell F_2^2
        else
          let y = tell F_1^3 in tell F_3^4
      in
        tell F_4^5
```

The correspondence Λ between labels in H_a and those in the code is here given: $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$.

We will present a more formal treatment after having introduced the typing rules for calculating both history expressions and labelling environments.

5.2 Typing rules

Here, we only give a logical presentation of our type and effect system. We assume that our Datalog is typed, i.e., that each predicate has a fixed arity and a type (see [34]). From here onwards, we simply assume that there exists a Datalog typing function γ that given a goal G returns a list of pairs (x, type-of-x), for all the variables x in G.

Typing environments The rules of our type and effect systems have:

- a standard environment Γ binding the variables of an expression:

  $$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

  where $\emptyset$ denotes the empty environment and $\Gamma, x : \tau$ denotes an environment having a binding for the variable x (x does not occur in Γ).

- a further environment K that maps a parameter $\tilde{x}$ to a pair consisting of a type and an abstract variation Δ. The information in Δ is used to resolve the binding for $\tilde{x}$ at runtime. Formally:

  $$K ::= \emptyset \mid K, (\tilde{x}, \tau, \Delta)$$

  where $\emptyset$ denotes the empty environment and $K, (\tilde{x}, \tau, \Delta)$ denotes an environment having a binding for the parameter $\tilde{x}$ ($\tilde{x}$ does not occur in K).

Judgements Our typing judgements have the form

$$\Gamma;\, K \vdash e \,:\, \tau \triangleright H;\, \Lambda$$

meaning that in the environments Γ and K the expression e has type τ, effect H and yields a labelling environment Λ.

Type syntax The syntax of types is

$$\tau_c \in \{int, bool, unit, \ldots\} \qquad \phi \in \wp(Fact)$$
$$\tau ::= \tau_c \mid \tau_1 \xrightarrow{K|H} \tau_2 \mid \tau_1 \xRightarrow{K|\Delta} \tau_2 \mid fact_\phi$$

We have basic types (*int*, *bool*, *unit*), functional types, behavioural variations types, and facts. Some types are annotated for analysis reason. In the type $fact_\phi$, the set ϕ soundly contains the facts that an expression can be reduced to at runtime (see the semantics rules (Tell2) and (Retract2)). In the type $\tau_1 \xrightarrow{K|H} \tau_2$ associated with a function f, the environment K is a precondition needed to apply f. Here, K stores the types and the abstract variations of parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e., the sequence of actions which may be performed over the context during the function evaluation. Analogously, in the type $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv and Δ is an abstract variation representing the information that the dispatching mechanism uses at runtime to apply bv.

Orderings We now introduce the *orderings* $\sqsubseteq_H, \sqsubseteq_\Delta, \sqsubseteq_K, \sqsubseteq_\Lambda$ on H, Δ, K and Λ, respectively (often omitting the indexes when unambiguous). We define:

- $H_1 \sqsubseteq_H H_2$ if and only if $\exists H_3$ such that $H_2 = H_1 + H_3$;
- $\Delta_1 \sqsubseteq_\Delta, \Delta_2$ if and only if $\exists \Delta_3$ such that $\Delta_2 = \Delta_1 \otimes \Delta_3$ (note that the concatenation Δ_2 has a single trailing term *fail*);
- $K_1 \sqsubseteq_K K_2$ if and only if $(\,(\tilde{x},\, \tau_1,\, \Delta_1) \in K_1$ implies $(\tilde{x},\, \tau_2,\, \Delta_2) \in K_2$ and $\tau_1 \leq \tau_2 \,\wedge\, \Delta_1 \sqsubseteq_\Delta \Delta_2\,)$;
- $\Lambda_1 \sqsubseteq_\Lambda \Lambda_2$ if and only if $\exists \Lambda_3$ such that $\Lambda_2 = \Lambda_1 \uplus \Lambda_3$.

We exploit this orderings to define rules for subtyping and subeffecting, as shown in Figure 5.2.

$$\frac{}{\tau \leq \tau}\ (\textsc{Srefl}) \qquad \frac{\phi \subseteq \phi'}{fact_\phi \leq fact_{\phi'}}\ (\textsc{Sfact}) \qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq K' \qquad H \sqsubseteq H'}{\tau_1 \xrightarrow{K|H} \tau_2 \leq \tau_1' \xrightarrow{K'|H'} \tau_2'}\ (\textsc{Sfun})$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq K' \qquad H \sqsubseteq H'}{\tau_1 \xRightarrow{K|H} \tau_2 \leq \tau_1' \xRightarrow{K'|H'} \tau_2'}\ (\textsc{Sva})$$

$$\frac{\Gamma;\, K \vdash e : \tau \triangleright H;\, \Lambda \qquad \tau \leq \tau' \qquad H \sqsubseteq H' \qquad \Lambda \sqsubseteq \Lambda'}{\Gamma;\, K \vdash e : \tau' \triangleright H';\, \Lambda'}\ (\textsc{Tsub})$$

Figure 5.2: Rules for subtyping and subeffecting

Most of the rules of our type and effect system are inherited from those of ML and properly extended (see Figure 5.3); those for the new constructs are in Figure 5.4. To make the presentation simpler, the labels (both of the expressions and of the history expressions) are explicited only in those rules that introduce new associations to the labelling environment. A few comments are in order.

Subtyping and subeffecting We have rules for subtyping and sub-effecting (displayed in Figure 5.2). As expected these rules say that subtyping relation is reflexive (rule (Srefl)); a type $fact_\phi$ is a subtype of a type $fact_{\phi'}$ whenever $\phi \subseteq \phi'$ (rule (Sfact)); functional types are contravariant in the types of arguments and covariant in the result type and in the annotations (rule (Sfun)); analogously for behavioural variations types (rule (Sva)). The rule (Tsub) allows us to freely enlarge types and effects by applying the subtyping and subeffecting rules; also, we can add elements to Λ, provided that there is no clash.

(TCONST)
$$\frac{v \in Val}{\Gamma;\, K \vdash v : \tau_v \triangleright \epsilon;\, \bot}$$

(TVAR)
$$\frac{\Gamma(x) = \tau}{\Gamma;\, K \vdash x : \tau \triangleright \epsilon;\, \bot}$$

(TIF)
$$\frac{\Gamma;\, K \vdash e_1 : \mathit{bool} \triangleright H_1;\, \Lambda \qquad \Gamma;\, K \vdash e_2 : \tau \triangleright H_2;\, \Lambda \qquad \Gamma;\, K \vdash e_3 : \tau \triangleright H_3;\, \Lambda}{\Gamma;\, K \vdash \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \tau \triangleright H_1 \cdot (H_2 + H_3);\, \Lambda}$$

(TLET)
$$\frac{\Gamma;\, K \vdash e_1 : \tau_1 \triangleright H_1;\, \Lambda_1 \qquad \Gamma, x : \tau_1;\, K \vdash e_2 : \tau_2 \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2 \triangleright H_1 \cdot H_2;\, \Lambda_1 \uplus \Lambda_2}$$

(TABS)
$$\frac{\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K' \vdash e\ :\ \tau_2 \triangleright H;\, \Lambda}{\Gamma;\, K \vdash \lambda_f x.e\ :\ \tau_1 \xrightarrow{K'|H} \tau_2 \triangleright \epsilon;\, \Lambda}$$

(TAPP)
$$\frac{\Gamma;\, K \vdash e_1\ :\ \tau_1 \xrightarrow{K'|H_3} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2\ :\ \tau_1 \triangleright H_2;\, \Lambda_2 \qquad K' \sqsubseteq K}{\Gamma;\, K' \vdash e_1\, e_2\ :\ \tau_2 \triangleright H_1 \cdot H_2 \cdot H_3;\, \Lambda_1 \uplus \Lambda_2}$$

Figure 5.3: Typing rules for standard ML constructs

(TFACT)

$$\frac{}{\Gamma;\, K \vdash F \,:\, fact_{\{F\}} \triangleright \epsilon;\, \bot}$$

(TTELL)

$$\frac{\Gamma;\, K \vdash e \,:\, fact_{\phi} \triangleright H;\, \Lambda}{\Gamma;\, K \vdash tell(e)^{l} \,:\, unit \triangleright \left(H \cdot \left(\sum_{F_i \in \phi} tell\, F_i^{l_i} \right) \right)^{l'} ;\, \Lambda \biguplus_{F_i \in \phi} [l_i \mapsto l]}$$

(TRETRACT)

$$\frac{\Gamma;\, K \vdash e \,:\, fact_{\phi} \triangleright H;\, \Lambda}{\Gamma;\, K \vdash retract(e)^{l} \,:\, unit \triangleright \left(H \cdot \left(\sum_{F_i \in \phi} retract\, F_i^{l_i} \right) \right)^{l'} ;\, \Lambda \biguplus_{F_i \in \phi} [l_i \mapsto l]}$$

(TVARIATION)

$$\frac{\begin{array}{c} \forall i \in \{1, \ldots, n\} \\ \gamma(G_i) = \overrightarrow{y_i} \,:\, \overrightarrow{\tau_i} \qquad \Gamma, x : \tau_1, \overrightarrow{y_i} \,:\, \overrightarrow{\tau_i}; K' \vdash e_i \,:\, \tau_2 \triangleright H_i;, \Lambda_i \\ \Delta = ask\, G_1.H_1 \otimes \cdots \otimes ask\, G_n.H_n \otimes fail \end{array}}{\Gamma;\, K \vdash (x)\{G_1.e_1, \ldots, G_n.e_n\} \,:\, \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright \epsilon;\quad \biguplus_{i \in \{1,\ldots,n\}} \Lambda_i}$$

(TVAPP)

$$\frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2;\, \Lambda_2 \qquad K' \sqsubseteq K}{\Gamma;\, K \vdash \#(e_1, e_2) \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot \Delta;\, \Lambda_1 \uplus \Lambda_2}$$

(TAPPEND)

$$\frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xRightarrow{K'|\Delta_1} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \xRightarrow{K'|\Delta_2} \tau_2 \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash e_1 \cup e_2 \,:\, \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright H_1 \cdot H_2;\, \Lambda_1 \uplus \Lambda_2}$$

(TPAR)

$$\frac{K(\tilde{x}) = (\tau,\, \Delta)}{\Gamma;\, K \vdash \tilde{x} \,:\, \tau \triangleright \Delta;\, \bot}$$

(TDLET)

$$\frac{\Gamma, \overrightarrow{y} \,:\, \overrightarrow{\tau};\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K, (\tilde{x},\, \tau_1,\, \Delta') \vdash e_2 \,:\, \tau \triangleright H;\, \Lambda_2}{\Gamma;\, K \vdash dlet\, \tilde{x} \,=\, e_1\, when\, G\, in\, e_2 \,:\, \tau \triangleright H;\, \Lambda_1 \uplus \Lambda_2}$$

where $\gamma(G) = \overrightarrow{y} : \overrightarrow{\tau}$
if $K(\tilde{x}) = (\tau_1,\, \Delta)$ then $\Delta' = G.H_1 \otimes \Delta$
else (if $\tilde{x} \notin K$ then $\Delta' = G.H_1 \otimes fail$)

Figure 5.4: Typing rules for new constructs

Type and effect of expressions Rules for standard ML constructs are shown in Figure 5.3 and need no comment. From now on, we focus on the rules for the constructs introduced in $\mathrm{ML}_{\mathrm{CoDa}}$(displayed in Figure 5.4).

The rule (TFACT) says that a fact F has type $fact$ annotated with the singleton $\{F\}$ and empty effect. The rule (TTELL)/(TRETRACT) asserts that the expression $tell(e)/retract(e)$ has type $unit$, provided that the type of e is $fact_\phi$. The overall effect is obtained by concatenating the effect of e with the nondeterministic summation of $tell\,F/retract\,F$ where F is any of the facts in the type of e.

In rule (TVARIATION) we determine the type for each subexpression e_i under K' and the environment Γ extended by the type of x and of the variables $\overrightarrow{y_i}$ occurring in the goal G_i (recall that the Datalog typing function γ returns a list of pairs (z, type-of-z) for all variable z of G_i). Note that all subexpressions e_i have the same type τ_2. We also require that the abstract variation Δ results from concatenating $ask\,G_i$ with the effect computed for e_i. The type of the behavioural variation is annotated by K' and Δ. Consider, e.g., the behavioural variation $bv_1 = (x)\{G_1.e_1, G_2.e_2\}$. Assume that the two cases of this behavioural variation have type τ and effects H_1 and H_2, respectively, under the environment $\Gamma, x : int$ (goals have no variables) and the guessed environment K'. Hence, the type of bv_1 will be $int \xRightarrow{K'|\Delta} \tau$ with $\Delta = ask\,G_1.H_1 \otimes ask\,G_2.H_2 \otimes fail$ and the effect will be empty.

The rule (TVAPP) type-checks behavioural variation applications and reveals the role of preconditions. As expected, e_1 is a behavioural variation with parameter of type τ_1 and e_2 has type τ_1. We get a type if the environment K', that acts as a precondition, is included in K according to $\sqsubseteq$. The type of the behavioural variation application is τ_2, i.e., the type of the result of e_1, and the effect is obtained by concatenating the ones of e_1 and e_2 with the history expression Δ, occurring in the annotation of the type of e_1. Consider, e.g., bv_1 above, its type and its empty effect ϵ. Assume to type-check $e = \#(bv_1, 10)$ in the environments Γ and K. If $K' \sqsubseteq K$, the type of e is τ and its effect is $\epsilon \cdot \Delta = ask\,G_1.H_1 \otimes ask\,G_2.H_2 \otimes fail$.

The rule (TAPPEND) asserts that two expressions e_1, e_2 with the same type τ, except for the abstract variations Δ_1, Δ_2 in their annotations, and effects H_1 and H_2, are combined into $e_1 \cup e_2$ with type τ, and concatenated annotations and effects. More precisely, the resulting annotation has the same precondition K' of e_1 and e_2 and abstract variation $\Delta_1 \otimes \Delta_2$, and effect $H_1 \cdot H_2$.

In rule (TPAR) we look for the type and the effect of the parameter $\tilde{x}$ in the environment K. The rule (TDLET) requires that e_1 has type τ_1 in the environment Γ extended with the types for the variables $\overrightarrow{y}$ of the goal G. Also, e_2 has to type-check in an environment K extended with the information for parameter $\tilde{x}$. The type and the effect for the overall $dlet$ expression are the same of e_2.

Handling the labelling environment The labelling environment generated by the rule (TCONST) is the empty one $\perp$. This is because no $tell$ or $retract$

l	1	2	3	4	5	6	7	8	9
$\Lambda(l)$	1	2	$\bot$	3	4	$\bot$	$\bot$	5	$\bot$

l	1	2	3	4	5	6	7	8	9
$\Lambda'(l)$	1	1	$\bot$	2	3	$\bot$	$\bot$	$\bot$	$\bot$

Figure 5.5: The labelling environment Λ for the expression e_a and its history expression H_a of Section 3 (on top); a non-injective environment Λ' (on the bottom).

occurs in the body of the expressions being typed, that as a matter of fact is just a value. The same happens for the rules (Tvar), (Tfact) and (Tpar), respectively for typing facts and parameters.

The most interesting rules are (Ttell) and (Tretract). They update the current environment Λ by associating all the labels of the facts which e can evaluate to, with the label l of the $tell(e)$ ($retract(e)$, resp.) being typed.

All the other rules just collect the associations created while typing the subexpressions. This is done either with widening or by explicitly merging the labelling environments. The first is the case of (Tif): here we force the expressions e_1, e_2 and e_3 to yield the same environment Λ (just like e_2 and e_3 are forced to have the same type τ). For the other case, consider as instance the rule (Tlet): it produces an environment Λ that contains all the correspondences of Λ_1 and Λ_2 coming from e_1 ed e_2; note that unicity of the labelling is guaranteed by the condition $dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset$.

As an example, Figure 5.5 (top) shows again the correspondence of the labels in the expression e_a and those of its history expression H_a of Chapter 3 (already recalled also in Section 5.1).

Note that a labelling environment needs not to be injective: this is because rules (Ttell) and (Tretract) can map different $l_i \in Lab_H$ into the same $l \in Lab_C$. This happens when the type $fact_\phi$ is annotated with a set ϕ consist of more than a single fact. Consider, e.g., the ambient Λ' in Figure 5.5 (bottom part), computed for

e'_a = `let x =`
`tell(if y then F`$_1$ `else F`$_2$`)`^{1}
`in`
$\leftarrow$ `F`$_5$`.retract F`$_8^2$`,`
$\leftarrow$ `F`$_3$`.retract F`$_4^3$

and for its history expression

$$H'_a = ((tell\, F_1^1 + tell\, F_2^2)^3 \cdot (ask\, F_5.retract\, F_8^4 \otimes (ask\, F_3.retract\, F_4^5 \otimes fail^6)^7)^8)^9$$

Here, the non-injectivity comes from the fact that the conditional expression `if y then F`$_1$ `else F`$_2$ can evaluate either to F_1 or F_2. Thus, the corresponding

`tell` construct (labelled with 1) has type $fact_{\{F_1,F_2\}}$ and is abstracted by the history expression $(tell\, F_1 + tell\, F_2)$.

5.3 Soundness

Our type and effect system is sound with respect to the operational semantics. Here, we recall the results presented in [19], adapting them to our present extension of the type and effect system. It is convenient to introduce the following technical definitions (we denote with $K_{\setminus \tilde{x}}$ the environment obtained by removing from K the association for $\tilde{x}$).

Definition 5.2 (Typing dynamic environment)**.** Given the type environments Γ and K, we say that the dynamic environment ρ has type K under Γ (in symbols $\Gamma \vdash \rho : K$) iff $dom(\rho) \subseteq dom(K)$ and $\forall \tilde{x} \in dom(\rho)$:

- $\rho(\tilde{x}) = G_1.e_1, \ldots, G_n.e_n$,
- $K(\tilde{x}) = (\tau, \Delta)$, and
- $\forall i \in \{1, \ldots, n\}\,.\ \gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ and $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K_{\setminus \tilde{x}} \vdash e_i : \tau' \triangleright H_i$ where $\tau' \leq \tau$ and $\bigotimes_{i\in\{1,\ldots,n\}} G_i.H_i \sqsubseteq \Delta$.

Definition 5.3. Given H_1, H_2 then $H_1 \preccurlyeq H_2$ iff one of the following cases holds

(a) $H_1 \sqsubseteq H_2$; (b) $H_2 = H_3 \cdot H_1$ for some H_3;

(c) $H_2 = \bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i \otimes fail \ \wedge\ H_1 = H_i,\ i \in [1..n]$.

Intuitively, the above definition formalises the fact that the history expression H_1 could be obtained from H_2 by evaluation.

The soundness of our type and effect system easily derives from the following standard results.

Theorem 5.3.1 (Preservation)**.** *Let e_s be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s and such that $\Gamma \vdash \rho : K$.*
If $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$ and $\rho \vdash C, e_s \to C', e'_s$ then $\Gamma; K \vdash e'_s : \tau \triangleright H'_s; \Lambda'_s$ for some $H'_s,\ \Lambda'_s$, where

- *there exists $\overline{H}$ such that $\overline{H} \cdot H'_s \preccurlyeq H_s$ and $C, \overline{H} \cdot H'_s \to^+ C', H'_s$, and*
- *$\Lambda'_s \sqsubseteq \Lambda_s$.*

Proof. See Appendix A. □

The Progress Theorem assumes that the effect H does not reach *fail*, i.e., that the dispatching mechanism succeeds at runtime. We take care of ensuring this property in Section 6 (we write $\rho \vdash C, e \nrightarrow$ to intend that there exists no transition outgoing from C, e; same for history expressions).

Theorem 5.3.2 (Progress)**.**
Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho : K$.
If $\rho \vdash C$, $e_s \not\twoheadrightarrow \ \wedge\ C$, $H_s \not\twoheadrightarrow^{+} C'$, fail then e_s is a value.

Proof. See [19], Appendix B.1. □

The following corollary ensures that the history expression obtained as an effect of e over-approximates the actions that may be performed over the context during the evaluation of e.

Corollary 5.3.3 (Over-approximation)**.** *Let e be a closed expression.*
If $\Gamma; K \vdash e : \tau \triangleright H; \Lambda_s \ \wedge\ \rho \vdash C$, $e \rightarrow^{\star} C'$, e', for some ρ such that $\Gamma \vdash \rho : K$, then there exists a sequence of transitions C, $H \rightarrow^{\star} C'$, H', for some H'.

Proof. See [19], Appendix B.1. □

Note that the type of e' is the same of e, because of Theorem 5.3.1. Furthermore, the labelling environment Λ'_s obtained by typing e' is included in Λ_s.

Chapter 6

Loading-time Analysis

Our execution model for ML_{CoDa} extends that of [19]: the compiler produces a quadruple $(C_p, e_p, H_p, \Lambda_p)$ composed by the application context, the object code, the history expression over-approximating the behaviour of e_p and the labelling environment associating labels of H_p with those in the code. Given $(C_p, e_p, H_p, \Lambda_p)$, at loading time the virtual machine performs:

- a *linking* phase, in which the virtual machine of ML_{CoDa} resolves system variables and constructs the initial context C (combining C_p and the system context); and
- a *verification* phase, in which a graph $\mathcal{G}$ describing the possible evolutions of C is built, starting from H_p.

We exploit $\mathcal{G}$ in order to (i) verify whether applications adapt to all evolutions of C, i.e., that all dispatching invocations will always succeed (only programs which pass this verification phase will be run), as done in [19]; and (ii) detect which `tell`/`retract` may lead to a violation of the system policy (see Section 7).

Technically, we compute $\mathcal{G}$ through a static analysis, specified in terms of Flow Logic [37]. Below, we describe the specification of our analysis, and we introduce the notion of viable history expressions. Intuitively, a history expression is viable for an initial context if the dispatching mechanism always succeeds.

To support the formal development, we assume that all bound variables occurring in a history expression are distinct. So we can define a function $\mathbb{K}$ mapping a variable h^l to the history expression $(\mu h.H_1^{l_1})^{l_2}$ that introduces it.

6.1 Analysis

The static approximation is represented by a pair $(\Sigma_\circ, \Sigma_\bullet)$, called *estimate* for H, with $\Sigma_\circ, \Sigma_\bullet$: $Lab \to \wp(Context \cup \{\bullet\})$ and where $\bullet$ is the distinguished "failure" context representing a dispatching failure. For each label l,

- the set $\Sigma_\circ(l)$ over-approximates the set of contexts that may arise before evaluating H^l (call it *pre-set*); while
- $\Sigma_\bullet(l)$ over-approximates the set of contexts that may result from the evaluation of H^l (call it *post-set*).

The analysis is specified in terms of a set of clauses that operate upon judgments in the form $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, where

$$\vDash \subseteq \mathcal{AE} \times \mathbb{H}$$

and $\mathcal{AE} = (Lab \to \wp(Context \cup \{\bullet\}))^2$ is the domain of the results of the analysis (called *analysis estimates*) and $\mathbb{H}$ the set of history expressions. The judgment $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$ expresses that $\Sigma_\circ$ and $\Sigma_\bullet$ constitute an acceptable analysis estimate for the history expression H^l.

The notion of acceptability will then be used in Definition 6.2 to check whether the history expression H_p, hence the expression e it is an abstraction of, will never fail in a given initial context C.

In Figure 6.1 we give the set of inference rules that validate the correctness of a given estimate, as presented in [19]. Now, we comment on them, where $\mathcal{E}$ denotes the estimate $(\Sigma_\circ, \Sigma_\bullet)$.

Intuitively, the estimate components take into account the possible dynamics of the language evaluation. The checks in the clauses mimic the semantic evolution of contexts, by modelling the semantic preconditions and the consequences of the possible reductions.

The rule (Anil) says that every pair of functions is an acceptable estimate for the "semantic" empty history expression $\mathfrak{d}$. The estimate $\mathcal{E}$ is acceptable for the "syntactic" ϵ^l if the pre-set is included in the post-set (rule (Aeps)). In the rule (Atell) the analysis checks whether the context C is in the pre-set, and the context $C \cup \{F\}$ is in the post-set; similarly for (Aretract), where $C \backslash \{F\}$ should be in the post-set. The rules (Aseq1) and (Aseq2) handle the sequential composition of history expressions. The rule (Aseq1) states that $(\Sigma_\circ, \Sigma_\bullet)$ is acceptable for $H = (H_1^{l_1} \cdot H_2^{l_2})^l$ if it is valid for both H_1 and H_2. Moreover, the pre-set of H_1 must include that of H and the pre-set of H_2 includes the post-set of H_1; finally, the post-set of H includes that of H_2. The rule (Aseq2) states that $\mathcal{E}$ is acceptable for $H = (\mathfrak{d} \cdot H_1^{l_2})^l$ if it is acceptable for H_1 and the pre-set of H_1 includes that of H, while the post-set of H includes that of H_1. By the rule (Asum), $\mathcal{E}$ is acceptable for $H = (H_1^{l_1} + H_2^{l_2})^l$ if it is valid for H_1 and H_2; the pre-set of H is included in the pre-sets of H_1 and H_2, and the post-set of H includes those of H_1 and H_2. The rules (Aask1) and (Aask2) handle the abstract dispatching mechanism. The first states that the estimate $\mathcal{E}$ is acceptable for $H = (askG.H_1^{l_1} \otimes \Delta^{l_2})^l$, provided that, for all C in the pre-set of H, if the goal G succeeds in C then the pre-set of H_1 includes that of H and the post-set of H includes that of H_1. Otherwise, the pre-set of Δ^{l_2} must include the one of H and the post-set of Δ^{l_2} is included in that of H. The rule (Aask2) requires $\bullet$ to be in the post-set of $fail$. By the rule (Arec) $\mathcal{E}$ is acceptable for $H = (\mu h.H_1^{l_1})^l$ if it is acceptable for $H_1^{l_1}$ and the pre-set of H_1

$$\frac{}{(\Sigma_\circ, \Sigma_\bullet) \vDash \mathfrak{d}} \text{(ANIL)} \qquad \frac{\Sigma_\circ(l) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash \epsilon^l} \text{(AEPS)}$$

$$\frac{\forall C \in \Sigma_\circ(l) \qquad C \cup \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash tell\, F^l} \text{(ATELL)} \qquad \frac{\forall C \in \Sigma_\circ(l) \qquad C \backslash \{F\} \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash retract\, F^l} \text{(ARETRACT)}$$

$$\frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1} \qquad (\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (H_1^{l_1} \cdot H_2^{l_2})^l} \text{(ASEQ1)}$$

$$\frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (\mathfrak{d} \cdot H_2^{l_2})^l} \text{(ASEQ2)}$$

$$\frac{\begin{array}{l}(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l) \\ (\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \vDash (H_1^{l_1} + H_2^{l_2})^l} \text{(ASUM)}$$

$$\frac{\begin{array}{l}\forall C \in \Sigma_\circ(l) \\ (C \vDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)) \\ (C \nvDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash \Delta^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l))\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \vDash (askG.H^{l_1} \otimes \Delta^{l_2})^l} \text{(ASK1)}$$

$$\frac{\bullet \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash fail^l} \text{(ASK2)} \qquad \frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (\mu h.H^{l_1})^l} \text{(AREC)}$$

$$\frac{\mathbb{K}(h) = (\mu h.H^{l_1})^{l'} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l') \qquad \Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash h^l} \text{(AVAR)}$$

Figure 6.1: Specification of the analysis for History Expressions

includes that of H and the post-set of H includes that of H_1. The rule (AVAR) says that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is an acceptable estimate for a variable h^l if the pre-set of the history expression introducing h, namely $\mathbb{K}(h)$, is included in that of h^l, and the post-set of h^l includes that of $\mathbb{K}(h)$.

6.2 From valid estimates to evolution graphs

We are now ready to introduce when an estimate for a history expression is valid for an initial context.

Definition 6.1 (Valid analysis estimate)**.** Given $H_p^{l_p}$ and an initial context C, we say that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is a *valid analysis estimate* for H_p and C iff $C \in \Sigma_\circ(l_p)$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash H_p^{l_p}$.

Semantic properties The following theorems state the correctness of our approach. The first guarantees that there exists a minimal valid analysis estimate, showing that the set of acceptable analyses forms a Moore family [37].

Theorem 6.2.1 (Existence of solutions)**.** *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

Proof. See [19], Appendix C. □

As expected, we have a standard subject reduction theorem, saying that the information recorded by a valid estimate is correct with respect to the operational semantics of history expressions.

Theorem 6.2.2 (Subject Reduction)**.** *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$.*
If for all $C \in \Sigma_\circ(l)$ it is $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H'^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.

Proof. See [19], Appendix C. □

Viability of history expressions We now define when a history expression H_p is viable for an initial context C, i.e., when it passes the verification phase. Below, let $lfail(H)$ be the set of labels of the *fail* sub-terms in H:

Definition 6.2 (Viability)**.** Let H_p be a history expression and C be an initial context. We say that H_p is *viable* for C if there exists the minimal valid analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ such that $\forall l \in dom(\Sigma_\bullet) \backslash lfail(H_P)$ it is $\bullet \notin \Sigma_\bullet(l)$.

We present now a couple of examples to illustrate how viability is checked. Consider the history expression

$$H_p = ((tell\ F_1^1 \cdot retract\ F_2^2)^3 + (ask\ F_5 . retract\ F_8^5 \otimes (ask\ F_3 . retract\ F_4^6 \otimes fail^7)^8)^4)^9$$

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
1	$\{\{F_2,\ F_5,\ F_8\}\}$	$\{\{F_1,\ F_2,\ F_5,\ F_8\}\}$
2	$\{\{F_1,\ F_2,\ F_5,\ F_8\}\}$	$\{\{F_1,\ F_5,\ F_8\}\}$
3	$\{\{F_2,\ F_5,\ F_8\}\}$	$\{\{F_1,\ F_5,\ F_8\}\}$
4	$\{\{F_2,\ F_5,\ F_8\}\}$	$\{\{F_2,\ F_5\}\}$
5	$\{\{F_2,\ F_5,\ F_8\}\}$	$\{\{F_2,\ F_5\}\}$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\{\bullet\}$
8	$\emptyset$	$\emptyset$
9	$\{\{F_2,\ F_5,\ F_8\}\}$	$\{\{F_1,\ F_5,\ F_8\},\{F_2,\ F_5\}\}$

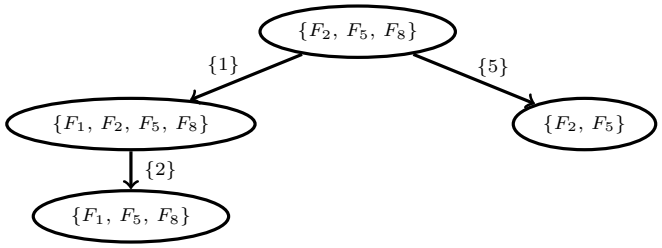

Figure 6.2: The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2,\ F_5,\ F_8\}$ and the history expression $H_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_5.retract\, F_8^5 \otimes ask\, F_3.retract\, F_4^6 \otimes fail^7)^4)^8$.

and the initial context $C = \{F_2,\ F_5,\ F_8\}$, consisting of facts only. For each label l occurring in H_p, Figure 6.2 shows the corresponding values of $\Sigma^1_\circ(l)$ and $\Sigma^1_\bullet(l)$, respectively. We can observe, e.g., that the pre-set for the *tell* labelled with 1 includes $\{F_2,\ F_5,\ F_8\}$, while the post-set includes $\{F_1, F_2,\ F_5,\ F_8\}$; similarly, the pre-set for the *remove* labelled with 5 includes $\{F_2,\ F_5,\ F_8\}$, while the post-set includes $\{F_2,\ F_5\}$. The column describing $\Sigma_\bullet$ contains $\bullet$ only for $l = 7$ which is the label of *fail*, so H_p is viable for C.

Now consider the following history expression that fails to pass the verification phase, when put in the same initial context C used above:

$$H'_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_3.retract\, F_4^5 \otimes fail^6)^4)^7$$

Indeed H'_p is not viable because the goal F_3 does not hold in C, and this is reflected by the occurrences of $\bullet$ in $\Sigma^2_\bullet(4)$ and $\Sigma^2_\bullet(7)$ as shown in Figure 6.3.

Now we exploit the result of the above analysis to build up the evolution graph $\mathcal{G}$ describing how the initial context C evolves at runtime, paving our way to security enforcement. Intuitively, $\mathcal{G}$ is a direct graph, the nodes of which are sets of contexts and there is an arc between two nodes C_1 and C_2 if C_2 is obtained from C_1 through adding or removing a fact F.

	$\Sigma^2_\circ$	$\Sigma^2_\bullet$
1	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
2	$\{\{F_1, F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
3	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
4	$\{\{F_2, F_5, F_8\}\}$	$\{\bullet\}$
5	$\emptyset$	$\emptyset$
6	$\{\{F_2, F_5, F_8\}\}$	$\{\bullet\}$
7	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}, \bullet\}$

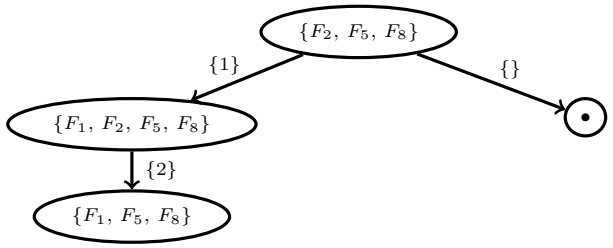

Figure 6.3: The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression $H'_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_3.retract\, F_4^5 \otimes fail^6)^4)^7$

In the following let $Fact^*$ and Lab^*_H be the set of facts and the set of labels occurring in H_p, the history expression under verification.

Definition 6.3 (Evolution Graph)**.** Let H_p be a history expression, C be an initial context, and $(\Sigma_\circ, \Sigma_\bullet)$ be a valid analysis estimate.
The evolution graph of C is $\mathcal{G} = (N, E, L)$, where

$$N = \bigcup_{l \in Lab^*_H} (\Sigma_\circ(l) \cup \Sigma_\bullet(l))$$

$$E = \{(C_1, C_2) \mid \exists F \in Fact^*,\ l \in Lab^*_H \text{ s.t. } C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge (h(l) \in \{tell(F), retract(F)\} \vee (C_2 = \bullet))\}$$

$$L : E \to \wp(Labels)$$

$$\forall t = (C_1, C_2) \in E,\ l \in L(t) \textit{ iff } C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge h(l) \neq fail$$

As examples of evolution graph consider the context C and the history expressions H_p and H'_p introduced in the examples above. The evolution graph of C for H_p is in Figure 6.2. From the initial context there is an arc with label 1 to the context $C \cup F_1$, because of the $tell^1$; also, there is an arc labelled 5 to the context without F_8, because of $retract^5$.

Clearly, the evolution graph $\mathcal{G}$ tell us when the dispatching mechanism always succeeds: it is sufficient to verify that the failure context $\bullet$ is not reachable from the initial context C. Back to our examples, it is easy to see that H_p is viable for C, because the node $\bullet$ is not reachable from C in the graph for H_p. Instead, H'_p is not viable, because $\bullet$ is reachable in the evolution graph for H'_p, displayed in Figure 6.3.

Note that labels of $\mathcal{G}$ also indicate which *tell*/*retract* may lead to a context violating the security policy Φ: in Chapter 7 we will exploit their correspondence with the labels in the code to enforce security checks.

Chapter 7

Code instrumentation

Now, we present the second component of our analysis. The previous chapter was about functional concerns of the language, that is, guaranteeing that a call to the dispatching mechanism never fails due to impossibility to adapt to the current context. Here we care abount security concerns: our efforts are aimed at preventing the application from violating the security policy Φ to obey. We reuse the information previouly collected, starting from the evolution graph and checking which reacheable contexts (if any) do not satisfy Φ; furthermore, we safely estimate which actions may lead to one of those contexts, and consequently instruct an ad-hoc runtime monitor. We proceed in two steps: the first one extracts new information from the graph; the second one uses it to monitor the execution steps. In the following, we formally describe our methodology.

7.1 A further static analysis step

We preliminarily detect which are the potential risky operations the application can perform through a static analysis of the evolution graph $\mathcal{G}$. The occurrence of these risky actions will then be guarded by our *runtime monitor*; on the others the monitor will be switched off.

Safe and unsafe transitions Since a node n of $\mathcal{G} = (N, E, L)$ represents a context that the application may reach during its execution, we first verify whether n satisfies Φ. We thus individuate the set of all the nodes corresponding to *forbidden* contexts

$$\mathcal{R}_N = \{n \in N \text{ such that } \Phi \text{ does not hold in } n\}$$

We then consider all the incoming edges of this nodes, building the set of the *risky* transitions

$$\mathcal{R}_E = \{t = (n_1, n_2) \mid n_2 \in \mathcal{R}_N\}$$

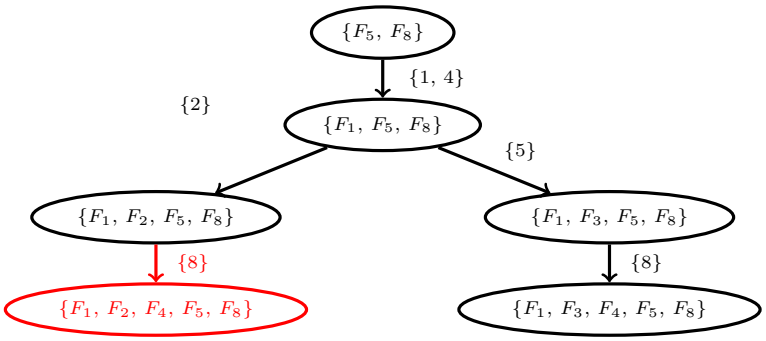

Figure 7.1: The evolution graph for the context $C \supseteq \{F_5, F_8\}$ and the history expression $H_a = (((tell\ F_1^1.tell\ F_2^2)^3 + (tell\ F_1^4.tell\ F_3^5)^6)^7.tell\ F_4^8)^9$ under policy $\Phi \equiv \neg F_2 \vee \neg F_4 \vee \neg F_8$. The (possible) sources of policy violation are highlighted in red, and represent the sets $\mathcal{R}_N$, $\mathcal{R}_E$, and $\mathcal{R}_L$.

and the set of their labels

$$\mathcal{R}_L = \bigcup_{t \in \mathcal{R}_E} L(t) \subseteq Lab_H$$

We eventually take advantage of the labelling environment Λ, computed while type checking the application, for building the set of those portions of the code that require to be monitored during the execution

$$Risky = \Lambda(\mathcal{R}_L) = \{\Lambda(l) \mid l \in \mathcal{R}_L\}$$

Clearly, if $\mathcal{R}_N$ results empty then also *Risky* will be empty, and we are sure that no policy violation will occur during the application execution. This is summerised by the following proposition.

Proposition 7.1.1 (Soundness of overapproximation)**.** *Consider an application having an evolution graph $\mathcal{G} = (N, E, L)$ and running under a security policy Φ. Let $\mathcal{R}_N \subseteq N$ be the set of the nodes that violate Φ, as discussed above. If $\mathcal{R}_N = \emptyset$ then also $Risky = \emptyset$, and no policy violation occurs during the execution of the application.*

Proof. Straightforward from the fact that the set N represents all the contexts possibly arising at runtime. □

Consider once more the evolution graph of Figure 7.1 in Chapter 3; suppose again that the security policy Φ requires $\neg F_2 \vee \neg F_4 \vee \neg F_8$, and that the labelling environment is $\Lambda \equiv \{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$. The leftmost node on the bottom (say $\bar{n}$, in red in the figure), that is, the context including the facts $\{F_1, F_2, F_4, F_5, F_8\}$ is the only one that violates Φ. Then, the set $\mathcal{R}_N$ consist only of the node $\bar{n}$. Since it has only one incoming edge, say $\bar{t}$, we have

$\mathcal{R}_E = \{\bar{t}\}$ and $\mathcal{R}_L = \{L(\bar{t})\} = \{8\}$. Applying the labelling environment, we finally find $Risky = \{\Lambda(8)\} = \{5\}$, which means that the only `tell/retract` that has to be guarded during the execution of the application is the one labelled by 5 inside the code.

Safe and unsafe states After having recognised the potential risky operations, we can detect (a subset of) the contexts that never raise violations and (a superset of) those who may. Indeed, being $\mathcal{G}$ a directed graph, there may exist some nodes from which we cannot reach any node in $\mathcal{R}_N$. Thus, we split the nodes of the graph in two classes: the ones from which there exists a path leading to a node in $\mathcal{R}_N$ and the others. If at runtime we reach a node of the last class, we are sure that a policy violation will never occur in the future.

Definition 7.1 (Red and blue nodes)**.** Given the graph $\mathcal{G} = (N, E, L)$ and the set $\mathcal{R}_N \subseteq N$, let $Red \subseteq N$ of the *unsafe* nodes be the smallest set satisfying the following conditions:

- $\mathcal{R}_n \subseteq Red$;
- if $t = (n_1, n_2) \in E$ and $n_2 \in Red$ then $n_1 \in Red$.

Conversely, we define the set of *safe* nodes as $Blue = N \setminus Red$.

The following observation immediately follows from the above definition.

Observation 1. Given a node $n \in N$, we have that if $n \in Red$ then there exists a path starting from n and leading to a node in $\mathcal{R}_N$. Moreover, if such a path exists then $n \in Red$.

If we go back to the graph (and the policy) presented in Figure 7.1, where $\mathcal{R}_N$ contains the leftmost node on the bottom, it is easy to distinguish its red and blue nodes. The resulting sets are shown in Figure 7.2 (the blue nodes are the two rightmost ones on the bottom). Here we do not explicit the labels of the edges, just because they are not relevant.

7.2 Handling the runtime monitor

Once the information about safe and unsafe nodes and transitions has been fully collected, we can present how our runtime monitor is implemented and switched on and off. Recall that the operations that may result in a policy violation are `tell/retract`, because they may modify the context. Therefore, we have to consider all of them and investigate their effects. In doing this we exploit information about which of them are risky and which are not, as specified by the static analysis discussed above.

A naive solution More in detail, we want the security policy Φ to be cheched immediately after a potential risky operation, that is, a `tell/retract` whose label belongs to the set $Risky$. To do that, the compiler has to provide specific

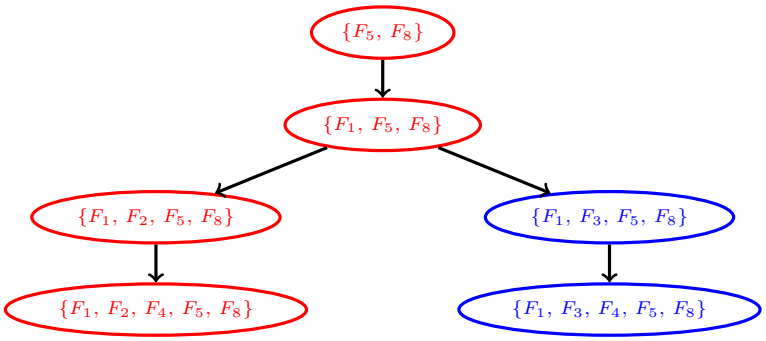

Figure 7.2: The sets *Red* and *Blue* for the evolution graph of Figure 7.1, under the same policy Φ.

support. Its actions can be summarised in the following two points: (*i*) labelling the source code as seen in Chapters 3 and 4 and (*ii*) generating specific calls to *trampoline-like* procedures. For now, assume as given a procedure for verifying whether the policy Φ is satisfied or not, say `check_whether_policy_violation(l)`, that takes a label `l` as parameter and has return type *unit*. A first, straightforward solution may be the following compilation schema: replace every $\texttt{tell(e)}^l$ in the source code with the following:

```
let z = tell(e) in
  check_whether_policy_violation(l)
```

where `z` is a fresh name; and do the analogous for every $\texttt{retract(e)}^l$:

```
let z = retract(e) in
  check_whether_policy_violation(l)
```

Note in passing that the so obtained code is again fully written in ML_{CoDa}, so we do not need any new mechanism to be introduced; moreover, this kind of translation is type-preserving: indeed the type of the `let` construct is that of `check_whether_policy_violation(l)`, which is *unit*, just like that of the `tell`. Last but not least, note that we have a lightweight form of code instrumentation that does not operate with the object code, differently from the standard instrumentations. Of course, we could have chosen to deal directly with the object code, but we preferred this approach, that works at a more abstract level and makes things simpler to deal with. In particular, our way is independent of the compilation schema and additionally it requires no modification to the compiler.

As said above, not all `tell`/`retract` expressions need the policy Φ to be checked again. The information gathered from the graph $\mathcal{G}$ and represented by the set *Risky* is used at linking time to set a global mask `risky[]`, assigned for

each label $\texttt{l} \in Lab_C$, as follows:

$$\texttt{risky[l]} = \begin{cases} \texttt{true} & \text{if } \texttt{l} \in \mathit{Risky} \\ \texttt{false} & \text{otherwise} \end{cases}$$

Now we can specify the procedure `check_whether_policy_violation`. Intuitively, it looks at `risky[l]`: if the value is `false` then no policy check is needed, the procedure returns to the caller and the execution goes on normally. Otherwise we have a call for a check on the policy Φ. The (pseudo) $\mathrm{ML}_{\mathrm{CoDa}}$ of this procedure is:

```
fun check_whether_policy_violation l =
  if risky[l] then
    ask phi.()
  else
    ()
```

Note that the call `ask phi.()` triggers a call to the dispatching mechanism that implements the check against the policy Φ: if the dispatching fails then a policy violation has been observed and the computation is aborted. This is indeed what we require to a runtime monitor: stop the application when a policy violation is about to occur.

A better solution The mechanism above induces a considerable overhead, introducing a procedure call each time we have an action that modifies the context. It is easy to speed it up, avoiding to invoke the procedure `check_whether_policy_violation` when the analysis of the evolution graph ensures that all the occurrences of `tell`/ `retract` are perfectly safe, because there is no execution path leading to a policy violation: this happens if and only if the set $\mathcal{R}_N$ (and, consequently, *Risky*) is empty. Still, a finer improvement is possible: whenever a `tell`/`retract` occurs leading to a new context $\tilde{C}$, one can look at the node $\tilde{n}$ of $\mathcal{G}$ corresponding to $\tilde{C}$. If $\bar{n} \in \mathit{Blue}$, then from now on no execution path leading to a policy violation exists, and the runtime monitor can be definitively switched off. To implement this optimization, we introduce the flag `always_ok`. Its value is initially computed at linking time, but it can be dinamically updated at runtime: once `always_ok` turns out to be true, no check is any longer needed and the monitor is switched off.

Then, we refine the previously compilation schema by testing the value of `always_ok` before calling our check procedure. All the occurrences $\texttt{tell(e)}^l$ in the source code are now replaced by

```
let z = tell(e) in
  if not(always_ok) then
    check_whether_policy_violation(l)
```

Similarly for the occurrences of $\texttt{retract(e)}^l$:

```
let z = retract(e) in
  if not(always_ok) then
    check_whether_policy_violation(l)
```

In this way, the execution time is likely to be reduced, because some costly, and useless security checks are not performed. Clearly, the bigger is the size of $Blue$ with respect to that of Red, the sooner (in expectation) the runtime monitor is switched off. It is reasonable to assume that execution paths leading to a policy violation are rare, which results in a cardinality of $Blue$ quite close to that of N and consequently in a (averagely) reduced overhead.

Chapter 8

A recovery strategy

The solution to the policy violation problem presented in Chapter 7 forces the application to stop its execution whenever it tries to make an action leading the context in a state with a violation. This strategy is clearly limiting, and its consequences can be unpleasant: one would like to provide the possibility of going back and taking different choices, so that the operativity of the application itself does not result fully compromised.

8.1 The need for a new mechanism

The language $\mathrm{ML}_{\mathrm{CoDa}}$ has two constucts that cause a branching in the program flow: they are *if-then-else* and (behavioural) variations. The conditional construct is guarded by a variable whose value is, generally, dynamically computed. We cannot predict or modify this value, so we cannot "artificially" force the execution of a branch rather than the other. But something can be done in case of behavioural variations. Recalling the semantics presented in Chapter 4, we only evaluate the expression corresponding to the leftmost satisfied goal. But what if other goals are satisfied? This is the general scenario: some expression are not evaluated even though their guard would allow for their execution. So, if the highest-priority expression (that is, the leftmost one) causes security problem, one can try to evaluate one of the others. From the user's point of view this proposal is acceptable: a behavioural variation is not indeed a construct for mutually exclusion, but just for providing adaptivity. In this way, we try to adapt the application not only to the context but to the security policy, too.

Consider, for instance, the expression $e = G_1.e_1,\ G_2.e_2,\ G_3.e_3$, and suppose that the goal G_2 is the only one that is not satisfied by the current context; suppose also that the expression e_1 is the only one leading to a policy violation. Being G_1 satisfied, the application would proceed executing e_1, and the runtime monitor would block it due to the policy violation. Then, a possible strategy may be testing the other goals: being G_3 also satisfied we would evaluate the expression e_3, allowing for a correct execution of the application.

Recalling again the example of the multimedia guide to a museum seen in Chapter 3, consider the function `getExhibitData()`: it may happen that the goal $\leftarrow$ `direct_comm()` is satisfied by the current context but for some reasons the call to `getChannel()` results in a policy violation, therefore not allowing for a direct communication through the available channels. Assuming that the goal $\leftarrow$ `use_qrcode(decoder), camera(cam)` is also satisfied, the application may proceed taking a picture of the QR code and decoding it. An alternative form of execution is then taken, which is both correct from the user's perspective and (hopefully) compatible with the security policy imposed by the system.

Try-catch vs behaviural variations An immediate strategy to handle the problem of security violation can be providing ML_{CoDa} with a *try-catch* construct. In this way, one would be able to catch the exceptions related to a policy violation and execute a different piece of code accordingly. This is also acceptable, but does not add any expressivity to the language if we assume already given the recovery strategy sketched above. Indeed, we have to distinguish two mutually exclusive cases: (i) the user has no knowledge of the system policy or (ii) the user knows a set $\mathcal{P}$ of security policies, from which the system chooses its current one (clearly, if $\mathcal{P}$ contains a single element then the user has full knowledge of the policy). In the first case, the code in the catch block cannot be parametric in the exception, because we do not know anything about it; then, this code could easily be expressed with a (piece of) behavioural variation, possibly guarding the resulting expression with a goal which is always true. Formally, the code `try` $\{Va\}$ `catch`$()\,\{e\}$ is the same as Va, $\leftarrow$ `true`$.\,e$. Suppose, as an example, that one wants to try to take a picture; in case this action results forbidden, he wants to switch off the flash and try again. Recalling the example presented in Chapter 3, this can be easily expressed as follows:

```
try {
  ← photocamera_started. tell button_clicked
} catch () {
  let x = tell photocamera_started in
  let y = retract flash_on in
  tell button_clicked
}
```

Thanks to the recovery mechanism, this is exactly the same as

```
  ← photocamera_started. tell button_clicked,
  ← true. let x =
       tell photocamera_started
    in
      let y = retract flash_on
    in
      tell button_clicked
```

In the other case, if we know that the current policy Φ comes from a fixed set $\mathcal{P}$, then we can collect information about Φ by just asking datalog goals. Thus, the code in the catch block can be now parametric in the type of the exception.

The same effect can be obtained with a behavioural variation setting the goals properly.

Finally, the user may want to specify a different catch block for each expression in the behavioural variation: the construct becomes $G_1.\texttt{try}\,\{e_1\}\,\texttt{catch}()\,\{e'_1\}, \ldots, G_n.\texttt{try}\,\{e_n\}\,\texttt{catch}()\,\{e'_n\}$. As an example of its usage, consider the following piece of code:

```
← photocamera_started.
            try {
              tell button_clicked
            } catch () {
              let y = retract flash_on in
              tell button_clicked
            } ,
← ¬ photocamera_available.
            /* do something else */
```

Here, the catch block is not common to all the expressions in the behavioural variation under consideration. Rather, it is specific of the expression guarded by the goal ← `photocamera_started`.

Again, this can be expressed in ML_{CoDa} without extending its syntax. We just need to repeat the goal, as follows: $G_1.e_1$, $G_1.e'_1, \ldots, G_n.e_n, G_n.e'_n$. The equivalence of the two expressions above is justified by the following intuition. If an expression e_i fails due to policy violation, then the recovery strategy immediately tries to execute e'_i, being its goal G_i obviously satisfied. Back to the example, the behavioural variation can therefore be written without using any try-catch construct, as follows:

```
← photocamera_started.
            tell button_clicked ,
← photocamera_started.
            let y = retract flash_on in
                tell button_clicked ,
← ¬ photocamera_available.
            /* do something else */
```

Thus, we can conclude that providing a try-catch construct would be just syntactic sugar, that could improve code readability.

8.2 The extended static analysis

If we adopt a recovery mechanism, our static analysis needs to be revised in order to (try to) guarantee new properties. Given a behavioural variation $Va = G_1.e_1, \ldots, G_n.e_n$ and a context $\bar{C}$, consider the set $\{e_i \mid \bar{C} \vDash G_i\}$ consisting of all the expressions having the guarding goal satisfied by $\bar{C}$. Consider now the evolution graph $\mathcal{G}$ built accordingly to the analysis presented in Chapter 6: the node $\bar{n}$ corresponding to the context $\bar{C}$ does not have an outgoing edge for each of the expression in the set above, but only one corresponding to the expression with the smallest index i. Instead, we want to consider also all these

missing edges, each one corresponding to an alternative path that one may follow whenever a recovery is needed.

8.2.1 The rules for the new analysis

To do this, the inference rules for valid analysis estimates need to be modified accordingly: in particular, the rule (Aask1) has to be rewritten so that one does not stop inspecting the (abstract) variation when a satisfied goal is found. As a consequence, to avoid introducing in the graph failure contexts $\bullet$ we distinguish two cases in which the analysis reaches $fail$: (i) the current context satisfies no goal, or (ii) all of the (satisfied) goals have been already considered. For this purpose, we introduce a new (labelled) history expression, say *stop*, similar to $fail$ but with a slighty different meaning. Then, instead of rule (Aask1) we use the new following rule:

(nAask1)

$$\frac{\begin{array}{c}\forall C \in \Sigma_\circ(l): \\ \big(C \vDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1} \\ (\Sigma_\circ, \Sigma_\bullet) \vDash \Delta^{l_2}[stop/fail] \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)\big) \\ (C \nvDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash \Delta^{l_2}) \\ \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)\end{array}}{(\Sigma_\circ, \Sigma_\bullet) \vDash (askG.H^{l_1} \otimes \Delta^{l_2})^l}$$

What we do is checking the first goal; if it is satisfied, we consider the remaining part with an additional information (the fact that $fail$ has been substituted with *stop*). Nothing new is done in case the first goal is not satisfied. Now, it is clear that if our analysis reaches $fail$ then no goal was satisfied, while reaching *stop* only means we have considered all the alternative paths. This last situation will be referred to with a dummy context $\bar{\bullet}$, whose eventual reachability obviously does non represent a failure. To manage this situation we introduce the following rule:

(nAask3)

$$\frac{\bar{\bullet} \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash stop^l}$$

To better understand how this mechanism works, in Figure 8.1 we show an example of analysis estimate and the corresponding evolution graph. To make things simpler, edges are not shown together with their actual label, but with the constructs they correspond to. Due to to the rule (nAask1), the abstract variation $(ask\, F_5.retract\, F_8^5 \otimes (ask\, F_2.retract\, F_5^6 \otimes fail^7)^8)^4)^9$ is now responsibile for more than one edge coming out from the initial context. Indeed, both the goals F_5 and F_2 are satisfied, so we have two egdes for $retract\, F_8^5$ and $retract\, F_5^6$ respectively. Note that the dotted edge corresponding to *stop* and

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
1	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
2	$\{\{F_1, F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
3	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\}\}$
4	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_5\},\{F_2, F_8\},\{\bar{\bullet}\}\}$
5	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_5\}\}$
6	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_8\}\}$
7	$\{\{F_2, F_5, F_8\}\}$	$\{\bar{\bullet}\}$
8	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_2, F_8\},\{\bar{\bullet}\}\}$
9	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\},\{F_2, F_5\},\{F_2, F_8\},\{\bar{\bullet}\}\}$

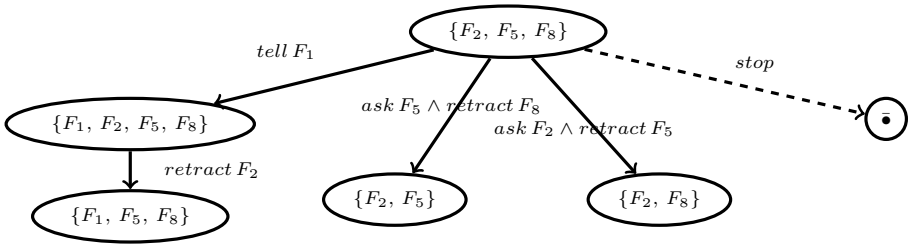

Figure 8.1: The analysis result (on top) and the evolution graph (on bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression $H_a = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_5.retract\, F_8^5 \otimes (ask\, F_2.retract\, F_5^6 \otimes fail^7)^8)^4)^9$.

leading to the dummy context $\bar{\bullet}$ does not really belong to the evolution graph (recall Definition 6.3: we only add edges corresponding to a *tell* or *retract*).

8.2.2 An ordering on the edges

Extending the analysis in the way above has an immediate consequence, that is, it may happen that some alternative paths that are now included lead to a functional failure $\bullet$. To better understand how this situation arises, consider the history expression $H_f = ask\, F_1.tell\, F_5 \otimes ask\, F_2.retract F_6 \otimes ask\, F_3.(retract\, F_8 \cdot (ask\, F_4.retract\, F_4 \otimes fail)) \otimes fail$ and an initial context $C = \{F_1, F_3, F_8\}$. In Figure 8.2 we compare the evolution graph obtained using the rules presented in Chapter 6 and the one obtained using the rules introduced above.

Here, a discussion is in order on the type of failures that the execution of an application may run into. We talk of *functional failure* when the dispatching mechanism gets stuck because no goal is satisfied, and of *security failure* if the computation is aborted due to a policy violation. If a functional failure occurs while executing an alternative path chosen by the recovery mechanism we just consider it as a security failure. Indeed, the failure occurs becuse the policy violation leads to take a different branch. From now on, we will refer to

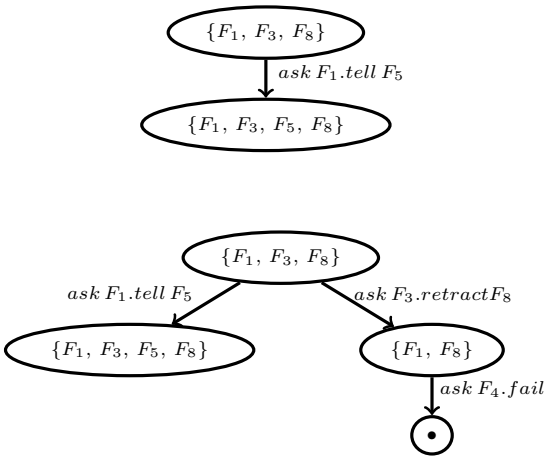

Figure 8.2: Two different graphs for the context $C = \{F_1, F_3, F_8\}$ and the history expression $H_f = ask\, F_1.tell\, F_5 \otimes ask\, F_2.retract F_6\ \otimes ask\, F_3.(retract\, F_8 \cdot (ask\, F_4.retract\, F_4 \otimes fail)) \otimes fail$.

execution paths without any intervention by the recovery mechanism as *standard paths*, while all the others are considered *alternative paths*; the same holds for paths in the evolution graph. Thus, a functional failure occurs if and only if a standard path get stuck; failures occuring in alternative paths will be all treated as security failures.

Clearly, we need a way to distinguish standard and alternative paths in the evolution graph, in order to better appoximate the corresponding real cases. A straightforward solution might be constructing the graph in two steps: the first using the old rules (that is, (Aask1) and (Aask2)), the other using the new rules ((nAask1) and (nAask3), together with (Aask2)). Edges coming from the first step are called *standard*, while the ones introduced by the second step are called *alternative*. However, all the information distinguishing between standard and alternative edges can be directly extracted from history expressions. To do this, we introduce a partial ordering $\prec$ on Lab_H.

Definition 8.1 (Ordering on labels)**.** Given an abstract variation $ask\, G_1.H_1$, $\ldots$, $ask\, G_n.H_n$, let $l(H_i)$ be the label of the first *tell* or *retract* occurring in H_i: then we impose $l(H_1) \prec \cdots \prec l(H_n)$. Given a history expression H, the ordering $\prec$ is defined as the union of all the orderings built as described above for the abstract variations occurring in H.

Note that the definition above requires the first *tell* or *retract* construct occurring in a history expression H_i to be uniquely defined. In general, this is not the case. It may happen that (i) H_i contains no *tell* or *retract* actions, or (ii) because of a non-deterministic construct, the first *tell* or *retract* is not well defined (consider, as instance, the history expression $tell\ F_1 + tell\ F_2$). To deal with

these situation, we require the compiler to add dummy *tell* constructs at the beginning of each history (sub)expression in abstract variations, each one telling a fresh, dummy fact that is relevant neither to the application behaviour nor to the security policies. Now, the ordering introduced above results well-defined.

Back to the evolution graph $\mathcal{G} = (N, E, L)$, consider now a labelled path $n_1 \xrightarrow{l_1} n_2 \xrightarrow{l_2} \cdots \xrightarrow{l_{k-1}} n_k$ where $t_i = (n_i, n_{i+1}) \in E$ and $l_i \in L(t_i)$. A labelled path is said to be *standard* (with respect to $\prec$) if and only if each l_i is a minimal element with respect to the ordering relation $\prec$, that is, there exists no label l such that $l \prec l_i, i = 1, ...k-1$. The other paths are said to be *alternative*.

This distinction enables us to deal with the failure nodes $\bullet$ in the graph, as follows:

Proposition 8.2.1. *Let H be a history expression, that abstracts the behaviour of the expression e. Let $\mathcal{G} = (N, E, L)$ be the evolution graph built for H starting from an initial context C. Let $\prec$ be the ordering induced by the history expression H. If there is no standard path (with respect to $\prec$) leading to $\bullet$ then no functional failure will occur during the evaluation of e.*

Proof. See Appendix A. □

8.3 Implementation issues

As usual, once the static analysis has been defined we have to show how the gathered information is used to implement new checks or features. For now ML_{CoDa} still has to be implemented, and this means that our recovery strategy can be treated only from an abstract point of view, not taking care about some technical details. In the following we illustrate some important points of out methodology, together with the problems that emerge and the way we faced them.

Checkpoints and recovery Implementing our recovery stragegy requires to get rid of some important issues. For now, we consider the *checkpoint* problem. Consider, as usual, a behavioural variation $Va = G_1.e_1, \ldots, G_n.e_n$ and suppose the evaluation of e_1 is aborted due to a policy violation. If we want to go back and try to evaluate one of the subsequent expressions, for example e_2 (assuming that G_2 is satisfied), we need to undo all the actions done by e_1 and restore the context as it was just before entering Va. This means that each time we enter a behavioural variation we have to keep a copy of the current context somewhere. Not only the context has to be saved, but also the global state, that is, the values of all the variables: as a consequence the size of the checkpoint could grow large.

Note in passing that this situation is very similar to that of transactions in databases: if a transaction is affected by a failure then the system has to be carried back to a consistent state, and this happens through *undo* or *redo* algorithms and the use of checkpoints.

Back to our problem, another issue we have to face is that behavioural variations can obviously be nested. So, if we want to be able to undo all of the actions done up to a certain moment, then a stack for checkpoints is needed, and its requirements in amount of memory may be very big. For this reason we limit our recovery strategy to the last behaviuoral variation the application has entered; when a new variation occurs, a new checkpoint is taken and the old one is discarded. Moreover, we are thinking of providing the programmer with a specific way to distinguish as *sensible* those variations he wants to be recovered in case of a security failure, if possible at all. Following this approach, a checkpoint is needed only when the application encounters a behavioural variation marked as sensible.

A guided smart recovery Finally, we briefly argue how our recovery strategy chooses an alternative path to follow. Our goal is avoiding as much as possible a new occurrence of a policy violation; so, among all the alternative paths, we would like to choose one, if any, that guarantees that no violation will be raised during its execution. To this aim, we are as usual supported by the results of our static analysis, and in particular by the evolution graph and by the ordering that has been defined on the labels of its edges.

Formally, let $Va = G_1.e_1, \ldots, G_n.e_n$ be the behavioural variation under consideration, H be its corresponding history expression, and suppose that the evaluation of Va begins in context C. Let n be the node of the evolution graph $\mathcal{G} = (N, E, L)$ representing the context C, and l be the label of the first `tell` or `retract` occurring in the reduction steps of H corresponding to the evaluation of Va. If a policy violation occurs, our recovery strategy has to choose an alternative path, whose first node is n and whose first edge has label l', with $l \prec l'$. Note that the ordering relation $\prec$ obviously includes the one induced by the history expression H of Va. Consider the set of the edges outgoing from n and corresponding to the beginning of an alternative path of Va

$$Out^E_{n,l} = \{t = (n, n') \in E \mid \exists l' \in L(t) \text{ with } l \prec l'\}$$

and the one of the nodes so connected to n

$$Out^N_{n,l} = \{n' \in N \mid (n, n') \in Out^E_{n,l}\}$$

Recall the set $Blue$ of safe nodes singled our by the static analysis. If the set $Out^N_{n,l} \cap Blue$ is not empty, then there is a *safe* alternative path, that is, an alternative path that guarantees the absence of policy violations in the rest of the execution. If this is the situation, our recovery strategy chooses a blue node n' in $Out^N_{n,l}$ and executes the alternative path starting with the edge (n, n'). It may happen that the set $Out^N_{n,l}$ contains more than one blue node: in this case we respect the priority specified by the user: among all the safe alternative paths, we choose the leftmost one.

The following example clarifies this strategy. Consider the history expression

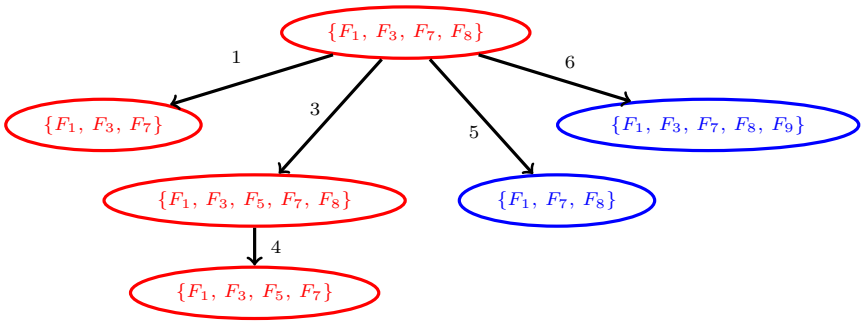

Figure 8.3: The sets *Red* and *Blue* in the evolution graph for the history expression $H_r = (ask\, F_1.retract\, F_8^1 \otimes (ask\, F_3.(tell\, F_5^3 \cdot retract\, F_8^4)^2 \otimes (ask\, F_7.retract\, F_3^5 \otimes (ask\, F_8.tell\, F_9^6 \otimes fail^7)^8)^9)^{10})^{11}$

H_r consisting of an abstract variation

$$\begin{aligned} H_r = &(ask\, F_1.retract\, F_8^1 \ \otimes \ (ask\, F_3.(tell\, F_5^3 \cdot retract\, F_8^4)^2 \otimes \\ &\otimes (ask\, F_7.retract\, F_3^5 \ \otimes \ (ask\, F_8.tell\, F_9^6 \ \otimes \ fail^7)^8)^9)^{10})^{11}. \end{aligned}$$

Suppose the initial context is $C = \{F_1, F_3, F_7, F_8\}$ and suppose also the security policy Φ requires that the fact F_8 always holds. The ordering $\prec$ induced by this abstract variation is $1 \prec 3 \prec 5 \prec 6$, while the resulting evolution graph is shown in Figure 8.3. The sets *Red* and *Blue* have been highlighted.
Standard evaluation would follow the leftmost edge (labelled by 1), resulting in a policy violation (the context $\{F_1, F_3, F_7\}$ does not contain the fact F_8, which is required by Φ). Said $\bar{n}$ the node corresponding to the initial context, the set $Out^E_{\bar{n},1}$ consists of the edges labelled by 3, 5 and 6 respectively, and $Out^N_{\bar{n},1}$ consists of the three rightmost sons of $\bar{n}$. Two of these nodes belong to the set *Blue*, thus the recovery strategy has to choose between them, selecting the leftmost one. Then, the edge labelled by 5 is followed, performing $retract\, F_3$ and reaching the (safe) context $\{F_1, F_7, F_8\}$.

Chapter 9

Conclusions

Following the Context-Oriented Programming paradigm, we considered the language for adaptive programming $\mathrm{ML}_{\mathrm{CoDa}}$ [20]. Here, in addition we addressed security issues, by suitably extending the two-phase static analysis for $\mathrm{ML}_{\mathrm{CoDa}}$ introduced in [19]. Our methodology and our main contributions can be summarised as follows:

- $\mathrm{ML}_{\mathrm{CoDa}}$, a core of ML extended with COP features, coupled with Datalog for dealing with contexts. We showed here that the Datalog component of the language suffices for expressing and for enforcing context-dependent security policies.

- A *type and effect system* for $\mathrm{ML}_{\mathrm{CoDa}}$ for ensuring that programs adequately respond to context changes, and for computing as effect an abstract representation of the overall behaviour. This representation, in the form of *history expressions*, abstractly describes the sequences of dynamic actions that a program may perform over the context. Our present extension also establishes a correspondence between the abstract actions in effects and the actual ones in the code, relevant to security.

- In [19] the effects are exploited at loading time to verify that the application can adapt to all contexts possibly arising at runtime. In addition, we built above a graph and a further *static analysis* that identifies the actions that may lead to contexts which violate the required *policy*. The analysis also detects some contexts that, once reached, guarantee the safeness of the future actions.

- We addressed security issues by defining a *runtime monitor* that stops an application when about to violate the policy to be enforced. The monitor exploits the link between the effects and the code. It is switched on and off, depending on the information collected by the static analysis mentioned above.

- Also, we designed a kind of *recovery* mechanism for behavioural variations, aimed at undoing some risky actions and taking different choices. This mechanism can be provided for each behavioural variation, or just in some portions of the code marked by the user as particularly sensitive. The recovery strategy is again guided by the results of the static analysis.

Still, our proposal is far from being complete and many improvements are possible, especially on the security side.

Future Work For now, $\mathrm{ML}_{\mathrm{CoDa}}$ still lacks an implementation. The design of a concrete version is one of the goals we want to achieve in the future. Once an implementation will be available, it will be possible to test how our metodology works in actual situations. This will help us in refining our analysis, if an improvement turns out to be necessary. Also, the type and effect system of $\mathrm{ML}_{\mathrm{CoDa}}$ provides the rules for type checking, but it is still missing a type inference algorithm. The realization of such an algorithm is another interesting issue.

As a future developement, we plan to extend the evolution graph with probabilities. This is aimed at making static estimates on the probability of encountering an execution path leading to a policy violation: if this value turns out to be higher than a threshold, then the application is prevented from running in that particular context. The threshold can be either a fixed value set by the system or specified by the user.

Another interesting direction of developement could be providing the language with a kind of machine learning, based on Bayesian inference. To do this, each action must be associated with a value that represents a reward or a penalty. Now, the evaluation of a behavioural variation does not depend on the first satisfied goal only, but has to consider all of the satisfied goals and to chose the expression that guarantees the best reward (in estimate). Of course the type and effect system needs to be revised in order to take into account these new features.

Acknowledgments

Part of the credit goes to my supervisors, that have consistently followed me and endured me since the early stages of research and writing that gave birth to this thesis. Besides them, there is a couple of people that I would like to thank for their helpful discussions and suggestions, and without whom this work would not have been the same. They are Gianluigi Ferrari (Università di Pisa) and Piero A. Bonatti (Università di Napoli). The first helped me when conceiving a thesis seemed tougher than I expected, and introduced me to $\mathrm{ML}_{\mathrm{CoDa}}$ for the very first time. The other spent some of his time to reccomend me some relevant papers about languages to express security policies and the expressivity of Stratified Datalog.

Bibliography

[1] Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: SOTA: Towards a General Model for Self-Adaptive Systems. 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises 0, 48–53 (2012)

[2] Achermann, F., Lumpe, M., Schneider, J., Nierstrasz, O.: PICCOLA—a small composition language. In: Formal methods for distributed processing. pp. 403–426. Cambridge University Press (2001)

[3] Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with java. Computer Software 28(1) (2011)

[4] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., Perscheid, M.: A comparison of context-oriented programming languages. In: International Workshop on Context-Oriented Programming (COP '09). pp. 6:1–6:6. ACM, New York, NY, USA (2009)

[5] Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and Verifying Service Composition. Journal of Computer Security 17(5), 799–837 (Oct 2009)

[6] Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6) (2009)

[7] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement Types for Secure Implementations. ACM Trans. Program. Lang. Syst. 33(2), 1–45 (Feb 2011)

[8] Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static Analysis for the *pi*-Calculus with Applications to Security. Information and Computation 168(1), 68 – 92 (2001)

[9] Bonatti, P., De Capitani Di Vimercati, S., Samarati, P.: An algebra for composing access control policies. ACM Transactions on Information and System Security 5(1), 1–35 (2002)

[10] Campbell, R., Al-Muhtadi, J., Naldurg, P., Sampemane, G., Mickunas, M.D.: Towards security and privacy for pervasive computing. In: Proc. of the 2002 Mext-NSF-JSPS international conference on Software security:

theories and systems (ISSS'02). Lecture Notes in Computer Science, vol. 2609, pp. 1–15. Springer-Verlag (2003)

[11] Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. on Knowl. and Data Eng. 1(1), 146–166 (1989)

[12] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)

[13] Costanza, P.: Language constructs for context-oriented programming. In: Proc. of the Dynamic Languages Symposium. pp. 1–10. ACM Press (2005)

[14] Costanza, P., Hirschfeld, R.: Language Constructs for Context-oriented Programming: An Overview of ContextL. In: Proceedings of the 2005 Symposium on Dynamic Languages. pp. 1–10. DLS '05, ACM, New York, NY, USA (2005)

[15] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POLP. pp. 238–252. ACM Press (1977)

[16] Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 269–282. POPL '79, ACM, New York, NY, USA (1979)

[17] DeTreville, J.: Binder, a Logic-Based Security Language. In: Proc. of the 2002 IEEE Symposium on Security and Privacy. pp. 105–113. SP '02, IEEE Computer Society, Washington, DC, USA (2002)

[18] Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems 5(1), 1–35 (1997)

[19] Galletta, L., Degano, P., Ferrari, G.L.: A staged static analysis for reliable adaptation, Submitted for publication - `http://www.cli.di.unipi.it/~galletta/staged_analysis.pdf`

[20] Galletta, L., Degano, P., Ferrari, G.L.: A two-component language for context oriented programming, Submitted for publication - `http://www.cli.di.unipi.it/~galletta/mlcoda.pdf`

[21] Ghezzi, C., Pradella, M., Salvaneschi, G.: An Evaluation of the Adaptation Capabilities in Programming Languages. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 50–59. SEAMS '11, ACM, New York, NY, USA (2011)

[22] Heer, T., Garcia-Morchon, O., Hummen, R., Keoh, S., Kumar, S., Wehrle, K.: Security challenges in the IP-based internet of things. Wireless Personal Communications pp. 1–16 (2011)

[23] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008 7(3), 125–151 (2008)

[24] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (Mar 2008)

[25] Hulsebosch, R., Salden, A., Bargh, M., Ebben, P., Reitsma, J.: Context sensitive access control. In: Proc. of the tenth ACM symposium on Access control models and technologies. pp. 111–119. ACM (2005)

[26] Jhala, R., Majumdar, R.: Software Model Checking. ACM Comput. Surv. 41(4), 1–54 (Oct 2009)

[27] Kamina, T., Aotani, T., Masuhara, H.: Eventcj: a context-oriented programming language with declarative event-based context transition. In: Proc. of the tenth international conference on Aspect-oriented software development (AOSD '11). pp. 253–264. ACM, New York, NY, USA (2011)

[28] Kephart, J., Walsh, W.: An artificial intelligence perspective on autonomic computing policies. In: In Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks. POLICY 2004. pp. 3–12 (2004)

[29] Khedker, U., Sanyal, A., Karkare, B.: Data Flow Analysis: Theory and Practice. CRC Press, Inc., Boca Raton, FL, USA (2009)

[30] Kokar, M., Baclawski, K., Eracar, Y.: Control theory-based foundations of self-controlling software. Intelligent Systems and their Applications, IEEE 14(3), 37–45 (1999)

[31] Kozen, D.: Language-based security. In: Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science. pp. 284–298. MFCS '99, Springer-Verlag, London, UK, UK (1999)

[32] Li, N., Mitchell, J.C.: DATALOG with Constraints: A Foundation for Trust Management Languages. In: Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages. pp. 58–73. PADL '03, Springer-Verlag, London, UK, UK (2003)

[33] Loke, S.W.: Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. Knowl. Eng. Rev. 19(3), 213–233 (2004)

[34] Mycroft, A., O'Keefe, R.A.: A polymorphic type system for prolog. Artificial Intelligence 23(3), 295 – 307 (1984)

[35] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, 1st ed. 1999. corr. 2nd printing, 1999 edn. (2005)

[36] Nielson, F., Nielson, H.: Type and effect systems. In: Olderog, E.R., Steffen, B. (eds.) Correct System Design, Lecture Notes in Computer Science, vol. 1710, pp. 114–136. Springer Berlin Heidelberg (1999)

[37] Nielson, H.R., Nielson, F.: Flow logic: a multi-paradigmatic approach to static analysis. In: Mogensen, T.A., Schmidt, D.A., Sudborough, I.H. (eds.) The essence of computation. Lecture Notes in Computer Science, vol. 2566, pp. 223–244. Springer-Verlag (2002)

[38] Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3), 54–62 (May 1999)

[39] Orsi, G., Tanca, L.: Context modelling and context-aware querying. In: Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog Reloaded, Lecture Notes in Computer Science, vol. 6702, pp. 225–244. Springer (2011)

[40] Pfleeger, C., Pfleeger, S.: Security in computing. Prentice Hall (2003)

[41] Puviani, M., Cabri, G., Zambonelli, F.: A Taxonomy of Architectural Patterns for Self-adaptive Systems. In: Proceedings of the International C* Conference on Computer Science and Software Engineering. pp. 77–85. C3S2E '13, ACM, New York, NY, USA (2013)

[42] Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., Nahrstedt, K.: Gaia: a middleware platform for active spaces. ACM SIGMOBILE Mobile Computing and Communications Review 6(4), 65–67 (2002)

[43] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 14:1–14:42 (2009)

[44] Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: A software engineering perspective. Journal of Systems and Software 85(8), 1801–1817 (2012)

[45] Schneider, F.B., Morrisett, G., Harper, R.: A language-based approach to security. In: Wilhelm, R. (ed.) Informatics, Lecture Notes in Computer Science, vol. 2000, pp. 86–101. Springer Berlin Heidelberg (2001)

[46] Skalka, C., Smith, S., Horn, D.V.: Types and trace effects of higher order programs. Journal of Functional Programming 18(2), 179–249 (2008)

[47] Wrona, K., Gomez, L.: Context-aware security and secure context-awareness in ubiquitous computing environments. In: XXI Autumn Meeting of Polish Information Processing Society (2005)

[48] Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 214–227. POPL '99, ACM, New York, NY, USA (1999)

[49] Zhang, G., Parashar, M.: Dynamic context-aware access control for grid applications. In: Proc. of Fourth International Workshop on Grid Computing, 2003. pp. 101–108. IEEE (2003)

Appendix A

Proofs

A.1 Theorems of Chapter 5

We need the following lemmas, inherited from those in [19].

Lemma A.1.1 (Substitution)**.** *If* $\Gamma, x : \tau'; K \vdash e \ : \ \tau \triangleright H; \ \Lambda$ *and* $\Gamma; K \vdash v \ : \ \tau' \triangleright \epsilon; \ \bot$ *then* $\Gamma, x : \tau'; K \vdash e\{v/x\} \ : \ \tau \triangleright H$*.*

Proof. By induction on the depth of the typing derivation and then by cases on the last rule applied. Omitted for the sake of brevity. □

Lemma A.1.2 (Restriction)**.** *If* $\Gamma; K \vdash v \ : \ \tau \triangleright H; \ \Lambda$ *then* $\Gamma; K \vdash v \ : \ \tau \triangleright \epsilon; \ \bot$

Proof. Being v a value, in the typing derivation for $\Gamma; K \vdash v \ : \ \tau \triangleright H; \ \Lambda$ there must be a subderivation with conclusion $\Gamma; K \vdash v \ : \ \tau' \triangleright \epsilon; \ \bot$ (rule (Tconst)) for some $\tau' \leq \tau$. Applying rule (Tsub), we can enlarge only the type but not the effect and the labelling environment: then we get $\Gamma; K \vdash v \ : \ \tau \triangleright \epsilon; \ \bot$. □

We are now ready to prove the following theorem.

Theorem A.1.3 (Preservation)**.** *Let* e_s *be a closed expression; and let* ρ *be a dynamic environment such that* $dom(\rho)$ *includes the set of parameters of* e_s *and such that* $\Gamma \vdash \rho \ : \ K$*.*
If $\Gamma; \ K \vdash e_s \ : \ \tau \triangleright H_s; \ \Lambda_s$ *and* $\rho \vdash C, \ e_s \rightarrow C', \ e'_s$ *then* $\Gamma; \ K \vdash e'_s \ : \ \tau \triangleright H'_s; \ \Lambda'_s$ *for some* $H'_s, \ \Lambda'_s$*, where*

(*i*) *there exists* $\overline{H}$ *such that* $\overline{H} \cdot H'_s \preccurlyeq H_s$ *and* $C, \overline{H} \cdot H'_s \rightarrow^+ C', \ H'_s$*, and*

(*ii*) $\Lambda'_s \sqsubseteq \Lambda_s$*.*

Proof. Part (*i*) is proven in [19], Appendix B.1. Here we prove part (*ii*). Similarly to that of part (*i*), the proof is by induction on the depth of the typing derivation and then by cases on the last rule applied.

- Rule (TVAR)
  We know e_s is a variable, so it cannot be the case that $\rho \vdash C,\ e_s \to C',\ e'_s$. The theorem simply holds. The same happens for rules (TABS), (TFACT) and (TVARIATION), with the exception that e_s is now a value.

- Rule (TTELL)
  We have $e_s = tell(e')^l$ for some e', l. Furthermore, the premises tolds us that $\Gamma;\ K \vdash e' : fact_\phi \triangleright H;\ \Lambda$ with $\Lambda_s = \Lambda \uplus_{F_i \in \phi} [l_i \mapsto l]$ and $H_s = (H \cdot \sum_{F_i \in \phi} tell\ F_i^{l_i})^l$. There are two rules from which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.

  - Rule (TELL1)
    For the premises we have $\rho \vdash C,\ e' \to C',\ e''$, then $e'_s = tell(e'')$. By inductive hypotesis we have $\Gamma;\ K \vdash e'' : fact_\phi \triangleright H'';\ \Lambda''$ with $\Lambda'' \sqsubseteq \Lambda$. By rule TTELL it follows that $\Gamma;\ K \vdash e'_s : unit \triangleright H'_s;\ \Lambda'_s$ with $H'_s = (H'' \cdot \sum_{F_i \in \phi} tell\ F_i^{l_i})^l$ and $\Lambda'_s = \Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l]$. Since $\Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l] \sqsubseteq \Lambda \uplus_{F_i \in \phi} [l_i \mapsto l] = \Lambda_s$ the thesis follows.
  - Rule (TELL2)
    Now $e_s = tell(F)$ so $e'_s = ()$ and $C' = C \cup \{F\}$. It follows that $\Gamma;\ K \vdash () : unit \triangleright \epsilon;\ \bot$, and since $\bot \sqsubseteq \Lambda_s$ the thesis follows.

- Rule (TIF)
  Given $e_s = if\ e_1\ then\ e_2\ else\ e_3$, we know that $\Gamma;\ K \vdash e_1 : bool \triangleright H_1;\ \Lambda$, $\Gamma;\ K \vdash e_2 : \tau \triangleright H_2;\ \Lambda$ and $\Gamma;\ K \vdash e_2 : \tau \triangleright H_3;\ \Lambda$. The transition $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived from three rules.

  - Rule (IF1)
    We have that $e'_s = if\ e'_1\ then\ e_2\ else\ e_3$. By inductive hypotesis, $\Gamma;\ K \vdash e'_1 : bool \triangleright H'_1;\ \Lambda'$ with $\Lambda' \sqsubseteq \Lambda$. Applying (TSUB) we get $\Gamma;\ K \vdash e'_1 : bool \triangleright H'_1;\ \Lambda$; by rule (TIF) it follows that $\Gamma;\ K \vdash e'_s : \tau \triangleright H'_1 \cdot (H_2 + H_3);\ \Lambda$. The thesis reduces to $\Lambda \sqsubseteq \Lambda$, which obviously holds.
  - Rule (IF2)
    In this case we have $e_1 = true$ and $e'_s = e_2$. Then $\Gamma;\ K \vdash e'_s : \tau \triangleright H_2;\ \Lambda$, and again the thesis reduces to $\Lambda \sqsubseteq \Lambda$.
  - Rule (IF3)
    Same as for (IF2), with the exception that $e_1 = false$ and $e's = e_1$.

- Rule (TLET)
  We have $e_s = let\ x = e_1\ in\ e_2$, and we know that $\Gamma;\ K \vdash e_1 : \tau \triangleright H_1;\ \Lambda_1$ and $\Gamma,\ x : \tau_1;\ K \vdash e_2 : \tau_2 \triangleright H_2;\ \Lambda_2$. There are two rules from which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.

  - Rule (LET1)
    Here $e'_s = let\ x = e'_1\ in\ e_2$. By inductive hypotesis, $\Gamma;\ K \vdash e'_1 : \tau_1 \triangleright H'_1;\ \Lambda'_1$ with $\Lambda'_1 \sqsubseteq \Lambda_1$. By rule (TLET) we get $\Gamma;\ K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2;\ \Lambda'_1 \uplus \Lambda_2$. Since $\Lambda'_1 \uplus \Lambda_2 \sqsubseteq \Lambda_1 \uplus \Lambda_2$ the theorem holds.

– Rule (Let2)
We have $e'_s = e_2\{v/x\}$, and we also know that $\Gamma;\ K \vdash v : \tau_1 \triangleright H_1;\ \Lambda_1$. By Lemma A.1.2 $\Gamma;\ K \vdash v : \tau_1 \triangleright \epsilon;\ \bot$ and applying Lemma A.1.1 we get $\Gamma;\ K \vdash e_2\{v/x\} : \tau_2 \triangleright H_2;\ \Lambda_2$. The thesis reduces to $\Lambda_2 \sqsubseteq \Lambda_1 \uplus \Lambda_2$, which is obviously true.

- For the remaining rules, the proof is similar to that of rule (Tlet).

□

A.2 Theorems of Chapter 8

Proposition A.2.1. *Let H be a history expression, that abstracts the behaviour of the expression e. Let $\mathcal{G} = (N, E, L)$ be the evolution graph built for H starting from an initial context C. Let $\prec$ be the ordering induced by the history expression H. If there is no standard path (with respect to $\prec$) leading to $\bullet$ then no functional failure will occur during the evaluation of e.*

Proof. Let $\mathcal{G}'$ be the graph that would have been obtained using rule (Aask1) instead of (nAask1). Suppose that a functional failure may occur at runtime. Then, there must be a path in $\mathcal{G}'$ leading to $\bullet$. But $\mathcal{G}'$ is a subgraph of $\mathcal{G}$, and paths in $\mathcal{G}'$ are standard paths in $\mathcal{G}$ (their labels are minimal with respect to $\prec$). This means that there is a standard path in $\mathcal{G}$ which leads to $\bullet$, contradicting the hypotesis of the preposition. Then the thesis holds. □