View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Archivio della Ricerca - Università di Pisa

The final publication is available at IOS Press through http://dx.doi.org/10.3233/JCS-160551

Context-aware Security: Linguistic Mechanisms and Static Analysis [1]

Chiara Bodei, Pierpaolo Degano, Letterio Galletta, Francesco Salvatori

Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Abstract

Adaptive systems improve their efficiency by modifying their behaviour to respond to changes in their operational environment. Also, security must adapt to these changes and policy enforcement becomes dependent on the dynamic contexts. We study these issues within $\mathrm{ML}_{\mathrm{CoDa}}$, (the core of) an adaptive declarative language proposed recently. A main characteristic of $\mathrm{ML}_{\mathrm{CoDa}}$ is to have two components: a logical one for handling the context and a functional one for computing. We extend this language with security policies that are expressed in logical terms. They are of two different kinds: context and application policies. The first, unknown *a priori* to an application, protect the context from unwanted changes. The others protect the applications from malicious actions of the context, can be nested and can be activated and deactivated according to their scope. An execution step can only occur if all the policies in force hold, under the control of an execution monitor. Beneficial to this is a type and effect system, which safely approximates the behaviour of an application, and a further static analysis, based on the computed effect. The last analysis can only be carried on at load time, when the execution context is known, and it enables us to efficiently enforce the security policies on the code execution, by instrumenting applications. The monitor is thus implemented within $\mathrm{ML}_{\mathrm{CoDa}}$, and it is only activated on those policies that may be infringed, and switched off otherwise.

Keywords: security policy, context-awareness, static analysis, type and effect system, control flow analysis, code instrumentation

1. Introduction

If you are in an airport for a little while and you want to have just a quick look at your mailbox, you would like to connect without bothering with all the details of the wireless connection, yet you would like to do that in a secure manner. A hosting environment and an application programmed in an adaptive fashion will transparently connect you to the available server. So, you never need to change your settings and capabilities, nor to worry about the new context and the resources it provides you. Instead you are likely to be worried about malicious activities that the hosting environment may carry on, and vice versa. Nevertheless, adaptivity amplifies the difficulties of security provisioning, because these two features are tightly interwoven. Their combination requires addressing two aspects. First, security may reduce adaptivity, because it adds further constraints on the possible actions of software. Second, new highly dynamic security mechanisms are needed to scale up to adaptive software.

[1] Work partially supported by the MIUR-PRIN project Security Horizons and by Università di Pisa PRA_2016_64 Project *Through the fog*.

These are the problems we address here from a linguistic point of view, in particular within a static analysis approach. To analyse such highly dynamic applications with a standard technique one has to take care of all the contexts that will be visited. This is computationally expensive and may even be unfeasible, because new contexts may appear later on, e.g. a new wireless network at the airport. We propose a technique that addresses these difficulties by splitting the analysis in two parts. The first collects at compile time information about the behaviour of the application regardless of the running context, while the second specialises this information at load time when the context is fully known.

Context and Adaptivity. As intuitively anticipated above, today's software systems are expected to operate *every time* and *everywhere*. They have to cope with changing environments, and never compromise their intended behaviour and their non-functional requirements, typically security or quality of service. Therefore, languages need effective mechanisms to sense the changes in the operational environment in which the application is plugged in, i.e. the *context*, and to properly *adapt* to changes, with little or no user involvement. 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. Typically, a context includes different kinds of computationally accessible information coming from both outside (e.g. sensor values, available devices, code libraries offered by the environment), and from inside the application boundaries (e.g. its private resources, user profiles, etc.). There have been many different proposals to support dynamic adjustments and tuning of programs inside programming languages, e.g. [45,38,39,64,66,67,42]. However, they do not neatly separate the working environment from the application as done, e.g. in Context Oriented Programming (COP) [20]. The COP linguistic paradigm explicitly deals with contexts and provides programming adaptation mechanisms to support dynamic changes of behaviour, in reaction to changes in the context. In this paradigm software adaptation is programmed through *behavioural variations*, chunks of code that can be automatically selected depending on the current context hosting the application, so to dynamically modify its execution.

Security and Contexts. Security is a major concern in context-aware systems, as witnessed by the interest risen in the business world [44], because an activity can be carried on safely in one operating environment and become weak in another. This implies that one has to determine which information in the environment is relevant for the security of the application. Also, it is important to constantly tune the security policies accordingly to the changes of the relevant part of the environment. New security techniques are therefore in order, that have to both scale up and maintain the ability of software to adapt as much as possible. These two interrelated aspects have already been studied in the literature [68,14] that presents two ways of addressing it: *securing context-aware systems* and *context-aware security*.

Securing context-aware systems aims at rethinking the standard notions of confidentiality, integrity and availability [58] and at developing techniques for guaranteeing them in adaptive applications [68]. The main challenge is to understand how to get secure and trusted context information. Contexts may indeed contain 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. Also, one has to protect applications from portions of the context that may misbehave and forge deceptive data.

Context-aware security is dually concerned with the use of context information (identity, location, time and date, device type, reputation and so on) to dynamically improve and drive security decisions. Contextual information helps to overcome the "one-size-fits-all" security solutions. These now become more flexible because they can take into account the different situations in which a user is operating.

For example, consider the usual no flash photography policy in museums. While a standard security policy *never* allows people to use flash, a context-aware security one does *not* allow flashing *only* inside particular rooms — in other words, the last policy has a scope. Similarly, a company might allow a user to access a database from the office, but deny access if the user attempts to from home, without an explicit authorisation. Accordingly, controlling and enforcing security need not to be placed everywhere, but only where needed, depending on the context that make specific actions risky.

Yet, there is no unifying concept of security, because the two aspects above are often tackled separately. Indeed, mechanisms have been implemented at different levels of the infrastructure, in the middleware [60] or in the interaction protocols [29]. These mechanisms mostly address access control, often from a software engineering viewpoint [56]. Also, particular attention is being paid on the ways contextual policies are defined [48]. Ours is a first step towards developing a uniform and formal treatment of security and adaptation.

Our proposal. We study security *within* a linguistic approach to adaptivity, and we propose techniques for analysing and enforcing secure behaviour of adaptive applications since the early stages of software development. To investigate these issues, we chose the COP paradigm because it provides us with a neat framework, where contexts and applications are clearly defined and identified, even though strictly intertwined. Consequently, this separation reduces the complexity of the security analysis by first considering the application in isolation, and then by tailoring the obtained results to the running context.

Here we extend ML$_{\text{CoDa}}$, a core of ML with COP features, recently proposed in [21,22,23]. It has two tightly integrated components: a declarative constituent for programming the context and a functional one for computing. The bipartite structure of ML$_{\text{CoDa}}$ reflects the separation of concerns between the specific abstractions for describing a context and those used for programming applications [61]. The context in ML$_{\text{CoDa}}$ is a knowledge base implemented as a (stratified, with negation) Datalog program [55,43]. Applications inspect the contents of a context by simply querying it, in spite of possibly complex deductions required. Programming adaptation is specified through *behavioural variations*, a sort of pattern matching with Datalog goals as selectors. The behavioural variation to be run is selected by the *dispatching* mechanism that inspects the actual context and makes the right choices. Note that the choice depends on both the application code and the "open" context, unknown at development time. If no alternative is viable, then a *functional failure* occurs as the application cannot adapt to the current context. We address context-aware security issues, in particular for defining and enforcing access control policies, by exploiting ML$_{\text{CoDa}}$ features. Our policies are expressed in Datalog, and are checked and enforced by just querying goals. In this regard, our version of Datalog is an asset because many logical languages for defining access control policies compile in it, e.g. [12,41,26]. In addition, it is powerful enough to express all relational algebras, it is fully decidable, and it guarantees polynomial response time [27]. There are two kinds of policies separately imposed by the context and by the applications. A single reference monitor enforces both at run time, aborting the execution when a security violation is about to occur.

More in detail, the designer of the context can define a *context policy* to protect some sensible entities hosted therein, e.g. specific devices or confidential data. The reference monitor prevents then an application running in that context from altering the values of the protected entities. Context policies are enforced along the execution of the application within the current context. Actually, the reference monitor is not required to stepwise supervise the execution of the controlled application, rather it only intervenes when an assertion concerning a protected entity changes in the context.

Instead, *application policies* are defined by the designer of the application, to protect its own resources and data. Following [9], we extend the original ML$_{\text{CoDa}}$ with the construct $\psi[e]$, called *policy framing*, where ψ is one of such policies and e is an expression within the application itself. The intuition is that the policy ψ has to be enforced stepwise during the execution of e, which is therefore its scope; when e is reduced to a value, the policy is no longer active. For example, suppose e is sending some confidential data. Then a policy can check whether all the recording devices in the context are off, so this sort of de-classification is not risky. When the transmission is completed, a recorder can be safely switched on, if the rest of the application does not care about. Of course, application policies can be nested, and are to be all obeyed by an expression occurring within the (nested) scope of their framing.

Observe that context and application policies have a different nature, mainly because they are defined by separate designers, in a completely independent manner. Actually, context policies are most likely unknown to the application developer at design and implementation time, and similarly for the application policies that are unknown to the context manager.

The execution model underlying our proposal assumes that the context is the interface between an application and the system running it. Applications interact with the system through a predefined set of APIs that provide handles to resources and operations on them. The system and the application do not trust each other, and may act maliciously. For instance, the first can alter some parts of the context inserting specific assertions so forcing the application to select a particular behavioural variation. The system can then falsify these assertions to drive the application in an unsafe state. The application developer can design and enforce a policy to protect sensible data against these malicious changes. In turn, the application can modify the context arbitrarily by driving the system in a vulnerable state, and context policies prevent these attacks. As a matter of fact, our policies specify the acceptable runs with respect to access control, so they are safety policies.

Here we do not address how code is protected against hostile modifications, and for that we assume our execution model to rely on known techniques, e.g. obfuscation [16]. A key point of our proposal is the instrumentation of the application code by inserting invocations to a reference monitor, and the resulting machinery is assumed to be suitably protected.

We aim at detecting policy violations (*non-functional failures*) as early as possible, so we conservatively extend the static approach of [28,22,23], briefly summarised below. It takes care of failures in adaptation to the current context (*functional failures*), dealing with the fact that applications operate in an "open" environment. Indeed, the actual value and even the presence of some elements in the current context are only known when the application is linked with it at run time. The first phase of our static analysis is based on a type and effect system that, at compile time, computes a safe over-approximation of the application behaviour, namely the *effect*. Then the effect is used at load time to verify that the resources required by the application are available in the actual context, and in its future modifications. To do that, the effect of the application is suitably combined with the effect of the APIs provided by the context that are computed by the same type and effect system. If an application passes this analysis, then no functional failure can arise at run time.

Our extensions are as follows. First, we record in the approximations the operations that modify the context, namely **`tell`** and **`retract`**, together with the scope of the policy framings in which they occur. The collected information is then used by our extended load time analysis. It requires building a graph, which safely approximates which contexts the application will pass through, while running. We also label the edges of the graph with (pointers to) the **`tell`**/**`retract`** operations in the code, exploiting the approximation. Before launching the execution, we detect the *risky* actions, i.e. those that might violate a reachable context or an active application policy. Now, we can call our reference monitor to guard the risky actions only, and leave it switched off for the remaining ones. Note that our two-phase analysis is similar to enforcing invariants over global data [52], which are however completely known at compile time. In our case instead the context is not available until run time, so an analysis at compile time would require predicting all the possible contexts an application will interact with — this is clearly overwhelming, or even unfeasible at all.

The detection of risky actions mentioned above drives code instrumentation, so to build applications that are prevented from violating security policies at run time. In a sense, the reference monitor is incorporated within the instrumented application, only using constructs of $\mathrm{ML}_{\mathrm{CoDa}}$ itself. Actually, the monitor is implemented by inserting in the source code of the application e a call to a suitable procedure (essentially a behavioural variation) for each occurrence of a **tell**/**retract** occurring in e. This form of instrumentation is not standard, as it does not operate on the object code, rather it is mechanically done at implementation time. Given a specific action and a set of policies, this procedure will obviously check whether the first obeys all the elements of its second argument. At load time, the information about which actions are risky and which policies can be violated by them is then used to link the actual and the formal parameters of the procedure. Indeed, the reference monitor is never called on those actions that the static analysis has safely predicted not to affect security.

Structure of the paper. The next section introduces $\mathrm{ML}_{\mathrm{CoDa}}$ and our proposal, with the help of an example, along with an intuitive presentation of the various components of our two-phase static analysis, and the way security is dynamically imposed and enforced. The syntax and the operational semantics of our extension of $\mathrm{ML}_{\mathrm{CoDa}}$ are in Section 3, and its type and effect system in Section 4. Section 5 presents the load time analysis, the results of which are used in Section 6 to describe our way of instrumenting applications. The conclusion summarises our results, discusses some future work, and briefly surveys related approaches. The proofs of the theorems establishing the correctness of our proposal are in the Appendix.

A preliminary version of some technical parts of this paper appeared in [11], that only considered context policies, and consequently had simpler definitions of the dynamic semantics, of the static analysis, and of the code instrumentation.

2. A guided tour of our proposal through an enterprise mobility scenario

We illustrate our proposal by considering a scenario of enterprise mobility, a typical example of ubiquitous and flexible computing: a mobile application used for accessing to some databases of a company through a tablet. For further details about the language and other applicative scenarios see [21,23].

We first intuitively introduce $\mathrm{ML}_{\mathrm{CoDa}}$, focussing on its logical aspects, used to inquire and update the context, and on our extensions to ML, used to program adaptation in a functional way. In particular, we will emphasise on how an application installs itself in a context (sub-section 2.1) and how the main adaptation feature, namely behavioural variation, is resolved through the dispatching mechanism (sub-section 2.2). In sub-section 2.3 we will add to our example both a context and an application policy, and discuss why at run time two different techniques are required for enforcing each of them. Then in sub-section 2.4, we exemplify two different ways an application may fail, either because it cannot adapt to the hosting context (functional failure) or because it or the context attempt to violate a policy (non functional failure). We also describe the two-phase static analysis we are proposing for detecting both kinds of failures and for efficiently controlling policies at run time. The first phase occurs at compile time and determines a sound abstraction of the behaviour of applications. The second phase analyses this abstraction at load time and provides us with the basis for instrumenting the code and for defining an adaptive reference monitor, that is only called on need. We conclude this intuitive presentation of our main contributions by briefly looking at some classical access control policies that show the expressivity of $\mathrm{ML}_{\mathrm{CoDa}}$.

2.1. Context Description and Updates.

As anticipated, the context in ML$_{\mathrm{CoDa}}$ is a knowledge base implemented as a (stratified, with negation) Datalog program [55,43]. To retrieve the state of a resource, programs simply query the context, in spite of the possibly complex deductions required to solve the corresponding goal; the context is changed by using the standard **`tell`**/**`retract`** constructs.

In our example, the usage of the tablet depends on the current location and on the profile of who is using it. For simplicity, we suppose that the tablet is able to recognise three locations each providing access to the network: (i) office, (ii) home, and (iii) public spots. Information on the current location can be retrieved by querying the context, described by a set of Datalog clauses. In our case, the user's location is described by the following clauses (for clarity, Datalog variables will be capital letters, while constants will be lower case identifiers):

```
location(office) ← wifi_connected(west_wing).
location(office) ← wifi_connected(east_wing).
location(home)   ← wifi_connected(myplace).
location(others) ← wifi_connected(X).
```

Assuming a mechanism that connects the tablet to a specific network, the predicate `wifi_connected` identifies the current operating environment by the network name. In particular, the tablet results to be in the office (the predicate `location(office)` holds) when connected to the network in either the west or east wing of the building.

The company employees can perform different actions depending on their profiles. We assume to have three of them: (i) vendor, (ii) system administrator, (iii) admin staff. The profile of a user is represented in the context by the binary fact `profile` (e.g. `profile(Jane, vendor)`). As expected, the user profile enables different applications and features: for example, vendors can access a database of customers, but not the company balance of payments; while a member of the staff can access both.

Furthermore, the context represents the resources currently available and manages the access to them in a declarative manner. Below we discuss some examples of different kinds of resources.

In the code snippet below, the first case returns a handle for accessing the contents of a digital archive that is stored in the office server, the second for the archive in a public server:

```
office_archive(db1) ← location(office),
                      available(office_server),
                      synchronised(office_server),
                      archive(office_server,db1).

office_archive(db2) ← location(others),
                      available(public_server),
                      synchronised(public_server),
                      archive(public_server,db2).
```

Of course, also physical or hybrid resources can be represented using Datalog, as the following typical situation arising in the Internet of Things [57]. Suppose that employees can remotely control the thermostat of their office room `R`, through the handle `S`, e.g. for checking whether their own is on or off.

```
 1 fun main () =
 2   dlet db_name = dbh when ← office_archive(dbh) in
 3   let records = (table){
 4     ← location(office),current_usr(name), profile(name,vendor).       // Goal G1
 5         tell accessing(db1)^1;                                        // Fact F1
 6         let c = open_db(db_name) in
 7           query(c, select * from table)
 8         retract accessing(db1)^2;
 9     ← location(others),current_usr(name),profile(name,vendor),
10        proxy(ip),crypto_key(k).                                       // Goal G2
11         let chan = connect(ip) in
12         tell accessing(db2)^3;                                        // Fact F2
13         let (c, sec_protocol) = get_db(chan) in
14         psi^4 [
15         let data = crypto_query(c, k, select * from table) in
16           decrypt(k, data)
17         ]
18         retract accessing(db2)^5
19   } in let result =  #(records, customers) in
20        display(result);
```

Figure 1. A simple ML$_{\text{CoDa}}$ application

```
thermostat(S) ← current_usr(U), usr_office(U,R), sensor(R,S).
```

In ML$_{\text{CoDa}}$ there are two manners for dynamically updating a context. The first exploits the API provided by the system, e.g. a function `disconnect` for closing the connection with a specific network. The second one is more explicit: a programmer can use the Datalog primitive constructs **`tell`** and **`retract`** that add and remove facts. In our example, a system administrator can change the profile of an employee, e.g. moving Bob to the vending department, as follows:

```
retract profile(Bob,admin_staff)
tell profile(Bob,vendor)
```

2.2. Behavioural adaptation.

The ML$_{\text{CoDa}}$ code in Figure 1 implements a simple application which accesses a database and performs a query to retrieve data about customers. The labels of the expressions **`tell`**, **`retract`**, and `psi` are not part of the code, and will be mechanically added for supporting the static analysis; they will be discussed later on and are to be ignored for the time being. The behaviour of the application depends on the location and on the profile of the user: when inside the office, the user can directly connect to and query the database. Otherwise, the communication exploits a proxy which allows getting the database handle.

A first example of specific ML$_{\text{CoDa}}$ construct, namely *context-dependent binding*, is at line 2 where the name for the office archive is taken from the context and bound to the variable `db_name`. The idea is to implement a simple form of adaptivity of data because the actual value is extracted from the current context, only known at run time. As a matter of fact, the value of `db_name` depends on the location of the user as prescribed by the definition of `office_archive`. The retrieved value is then used by `open_db` to open a connection to the database.

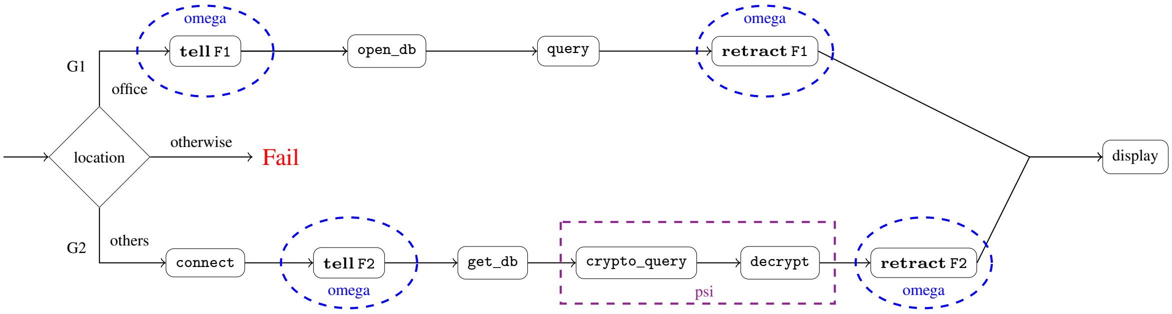

Figure 2. A flow chart intuitively describing the function `main`

The core of the snippet above is the behavioural variation (lines 3 - 18) bound to `records` that downloads the table of customers. Behavioural variations change the program flow in accordance to the current context. Syntactically, they are similar to pattern matching, where Datalog goals replace patterns and variables can additionally occur: a behavioural variation is a list $G_1.e_1, \ldots, G_n.e_n$ where the expressions e_i are guarded by Datalog goals G_i. In the code above, there are two alternatives, starting at lines 4 and 9 respectively. The selection between them depends on the location and on the capabilities of the current employee. To be more precise, behavioural variations are a sort of functional abstractions and their application triggers a specific *dispatching mechanism* that inspects the context and selects the expression e_i guarded by the first goal G_i satisfied. If no guards are satisfied, the computation gets stuck, as the application behaviour is undefined in that specific context: the application cannot adapt to it and a functional failure occurs.

In our example, the application behaves differently depending on the current location of the vendor, as intuitively shown in Figure 2. If the vendor is inside the office, he can directly access customers data connecting to `db1`; when outside, he has to first connect to the company proxy, then he can access `db2` through a secure channel; otherwise the application raises an error because running in an unexpected context. Note that every resource available to the application, is only accessible through a handle provided by the context and only manipulated through system functions provided by the API.

If the second case is selected, the IP address of an available proxy is retrieved by the predicate `proxy` that binds the corresponding handle to the variable `ip`. Then the application calls the API function `connect` to establish a communication through `chan`. By exploiting this channel the application gets a handle to the database (through the API function `get_db` at line 13) in order to obtain the required data. Accessing and releasing the relevant database is notified by updating the context through the **`tell`** and **`retract`** actions at lines 5, 12 and 8, 18, respectively. Note that the third argument of `crypto_query` at line 15 is a lambda expression (in a sugared syntax) that invokes another API function `select-from` (as common, we assume that the cryptographic primitives are supplied by the system). Other resources and APIs occur in the snippet above: the database connection channel `c` at line 6, the cryptographic key `k` at line 10, and `open_db` at line 6, `query` at line 7, `decrypt` at line 16.

The code lines 15 and 16 are within the scope of the application policy `psi`, discussed in the next sub-section. It guards the execution of the calls to `crypto_query` and `decrypt`: intuitively the policy `psi` guarantees the compatibility of the cryptographic key `k` and of the channel `c`.

2.3. Security policies.

We now discuss how the context protects itself against misuses by applications and symmetrically how applications constrain the usage of their data and resources. For that, ML$_{\mathrm{CoDa}}$ provides two kinds of security policies: *context* and *application* policies. As a matter of fact, policies predicate on the context, so they are easily expressed in Datalog and enforced by the deductive machinery of ML$_{\mathrm{CoDa}}$. As anticipated, the enforcement of a context policy ω is done by a reference monitor that checks the validity of ω right before every context change, i.e. before executing each **`tell`/`retract`**. Checking whether an application policy ψ is obeyed by a context has to be done continuously by the reference monitor, actually *before* and *after* each reduction step. Below we describe some examples of both kinds of policies.

Since the boundaries between personal and business space and time are blurred, the company adopts security policies to limit certain functionalities such as access to data from outside the office. In this scenario, the employees have different access rights: the vendors, among which Jane and Bob, can access the databases from both inside and outside the office, and the following facts specify which databases Bob and Jane can access, whereas the predicate allows an administrator to grant permissions:

```
has_auth(Bob,db1).
has_auth(Jane,db1).
has_auth(Jane,db2).
has_auth(X,D) ← delegate(Z,X,D), is_admin(Z).
```

A context policy that controls how vendors access the database `db2` from outside the office follows

```
omega ← current_usr(U), profile(U,vendor), has_auth(U,db2),
         location(others), accessing(db2).
```

Pictorially, the dotted circles in Figure 2 (blue in the pdf) represent that every modification of the context requires checking the policy `omega`.

As an example, if Jane is outside the office, no violation occurs. Instead, if Bob attempts to access the database from outside and has no delegation from the administrator, the policy `omega` is infringed at line 12. Of course policies can take into account any other kind of contextual information. One may e.g. constrain less the accesses of a vendor if he is using a corporate tablet whose running operating system is trusted. Also, a policy can prohibit vendors from accessing any databases during the week-ends.

As intuitively introduced above, the application policy `psi` regulates the usage of cryptographic keys and communication channels. We assume that the application has its own private key `k`, stored in the application context. The communication infrastructure offers channels supporting different cryptographic protocols. The application is designed in such a way that the key used is compatible with the protocol `sec_protocol` associated with the channel `c` returned by the call `get_db(chan)`. Context information is therefore used for choosing the suitable cryptographic protocol for the communication [40]. For simplicity, here compatibility means that the protocol can use keys of a certain length, following the context-agile encryption technique of [59]. Formally

```
psi ← length_key(k,X), protocol_supports(sec_protocol,X).
```

Note that an application policy cannot be rendered by a behavioural variation, because its guards are only checked when it is applied. Instead, the goal corresponding to the check of an application policy is queried by the reference monitor at each execution step. Assume in our simple example above that the protocol used to communicate and the channel `c` are compatible, but that the API `crypto_query` selects another protocol which is not compatible. Consequently, `psi` holds when entering the policy framing and a behavioural variation replacing line 14 will succeed at that execution point. However, a security violation occurs thereafter, because the case of the behavioural variation has been already selected. Instead our mechanism for application policy detects the violation: `psi` is indeed enforced stepwise along the execution of the API functions `crypto_query` (and `decrypt`) as intuitively shown by the dotted rectangle in Figure 2 (purple in the pdf).

2.4. Failures, Static Analysis and Instrumentation.

An application fails to adapt to a context (*functional failure*), when it has not been designed for the actual hosting context, e.g. because a missing facility was assumed to be present. In our example, this happens when a vendor attempts to access the database from home. A functional failure is reflected by a failure of the dispatching mechanism that causes the computation to get stuck. As a matter of fact, adaptive applications are prone to a new class of run time errors that are hard to catch, since the running contexts are unpredictable.

Another kind of failure happens when an application does not manipulate resources as expected (*non-functional failure*) and causes a violation of a policy. As said above, in our example Bob infringes the context policy `omega` when attempting to access the database `db2` if not delegated to. Another example is the violation of the application `psi` occurring when the key `k` is too short for the protocol associated with the channel `c` in lines 14-17.

To avoid functional failures and to optimise policy enforcement, we extend the two-phase static analysis of ML$_{\text{CoDa}}$ [22,23]. This analysis consists of (i) a type and effect system, and (ii) a control-flow analysis. It checks whether an application will be able to adapt to its execution contexts, and detects which contexts possibly violate the required policies.

In more detail, at *compile time*, we associate a type and an effect with an expression e. The type is (almost) standard, and the effect is an over-approximation of the actual run time behaviour of e, called *history expression*. The effect abstractly represents the changes and the queries performed on the context during its evaluation. The second phase occurs at *load time* and exploits the history expression to build a graph describing how the context may evolve during the execution.

For example, the history expression associated by the type system with the behavioural variation `records` is the following:

$$\begin{aligned} H_{\texttt{records}} = {} & ask\, G_1.\, tell\, F_1^{l_1} \cdot H_{\texttt{open\_db}} \cdot H_{\texttt{query}} \cdot retract\, F_1^{l_2} \otimes \\ & ask\, G_2.\, H_{\texttt{connect}} \cdot tell\, F_2^{l_3} \cdot H_{\texttt{get\_db}} \cdot \psi^{l_4}[H_{\texttt{crypto\_query}} \cdot H_{\texttt{decrypt}}] \cdot retract\, F_2^{l_5} \otimes \\ & fail \end{aligned}$$

In $H_{\texttt{records}}$ the symbol $\cdot$ abstracts sequential composition; ψ represents the application policy `psi`; $ask\, G_1. \cdots \otimes ask\, G_2. \cdots \otimes fail$ is the abstract counterpart of the behavioural variation `records`, where $\otimes$ sequentially composes the pair of effects associated with a guard and its expression; goals G_1 and G_2 represent the goals at lines 4 and 9, respectively; facts F_1 and F_2 are `accessing(db1)` and `accessing(db2)`.

History expressions are labelled (for the sake of readability, we just show the relevant ones in $H_{\texttt{records}}$). These labels enable us to link the abstract actions in histories to the corresponding actions of the code, that we assume labelled by the compiler. For instance, the $tell\ F_1^{l_1}$ in $H_{\texttt{records}}$ corresponds to the **tell** at line 5 in records, which is labelled by 1; the ψ^{l_4} is similarly linked to the policy framing with label 4. All the correspondences are $\{l_1 \mapsto 1, l_2 \mapsto 2, l_3 \mapsto 3, l_4 \mapsto 4, l_5 \mapsto 5\}$ (the abstract labels that do not annotate **tell/retract** actions or policies have no counterpart; also, the correspondence needs not to be injective as it happens in this example).

At *load time*, the virtual machine of ML$_{\text{CoDa}}$ performs two steps: linking and verification. The first step resolves the system names, and constructs the initial context C, by combining the one of the application with the system context that includes information on the current state, e.g. available resources and their usage constraints. The linking step also checks the logical consistency of the context C. The verification first checks whether no functional failure occurs, i.e. whether the application adapts to all evolutions of C that may occur at run time. If this is the case, the application will not be run. Then, the analysis looks for the points in the code where non-functional failures can occur, i.e. when the application may act against the policies established by the system that loads the program, and vice versa. This information is used to drive the activation of a run time monitor by need.

To perform our analysis conveniently and efficiently, we build a graph $\mathcal{G}$ describing the possible evolutions of the initial context C, through a control flow analysis of the history expression H. The nodes of $\mathcal{G}$ over-approximate the contexts arising at run time and its edges carry (the labels of) the actions which change the context. A distinguished aspect of our analysis is that it depends on the initial context C, right because our application may behave correctly in one context and fail in another, so this analysis can only be done at load time.

Back to our example, we consider three cases depending on different initial contexts, depicted in Figure 3, where for the sake of brevity we collapsed parts of the graph as a single edge labelled by the relevant history expression. The first initial context C_{home} records that the vendor is at home (predicate wifi_connected(myplace) holds). Since no case of the behavioural variation records can be selected because the application is not designed to work in that context. No guards are satisfied, and the dispatching mechanism fails. A pictorial representation of this functional failure is in Figure 3: the failure node $\circledast$ (red in the pdf) is reachable from the initial one. Suppose now that in the context C_{Bob} the predicate current_usr(Bob) holds. The $\texttt{tell}\,\mathrm{F}_2^{l_3}$ is risky because it may violate the context policy omega (actually it does), therefore the corresponding action, labelled by 3 in the code, must be blocked. For achieving this, it suffices switching on the run time monitor, right before executing this operation. Another potential non functional failure arises when the cryptographic key and the protocol are not compatible (the application policy psi is violated). This is shown in Figure 3 assuming as done above that the crypto_query may cause the violation. Again the correspondence between the label l_4 in the history expression and the label 4 in the code indicates that the run time monitor has to be switched on when entering the scope of the policy psi and switched off when leaving it. Of course, if in context C_1 one proves that the cryptographic key and the protocol agree, there is no potential non functional failure and the application runs without any monitoring.

2.5. Other examples of security policies.

We conclude the intuitive presentation of our proposal with a few examples showing how Datalog expresses other policies and how we manage them. In particular, we consider below delegation, dynamic activation and deactivation of policies, and a way of controlling which data in a context a user is allowed to access.

Imagine that the company allows a vendor to delegate another vendor to access the data of a particular customer without resorting to the system administrator as done above. Such a delegation is represented

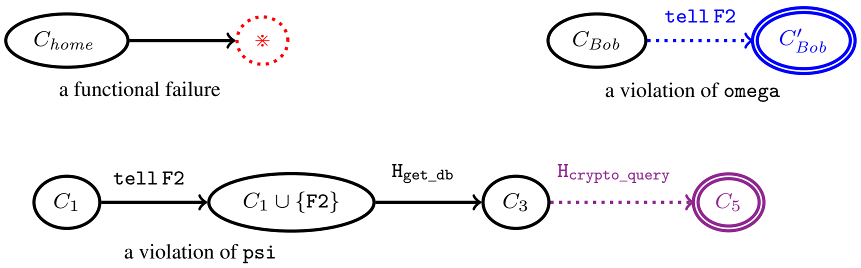

Figure 3. Three evolution graphs showing a functional failure (assuming that `wifi_connected(myplace)` holds in C_{home}) and two policy violations. The context policy `omega` is not obeyed because the user in C_{Bob} is Bob; the application policy `psi` is violated because the key is not compatible with the protocol. The dotted edges will be cut off by the run time monitor.

in the context as the fact `grant_permission(user1, user2)` meaning that the employee `user2` can operate in place of the employee `user1`. So a **`tell`** of this fact suffices to activate the delegation. Of course, the delegation is better constrained by the following policy that forbids a member of the staff to delegate a vendor and vice versa:

```
omega1 ← grant_permission(U1, U2), profile(U1, vendor), profile(U2, vendor).
```

In order to dynamically activate or deactivate context policies, we require that their definition includes a fact, that acts as an activation flag. If this fact holds in the current context, then the corresponding policy is active, otherwise it is not. So the context manager can activate/deactivate a policy by simply changing the value of the corresponding flag. For example, take the policy `omega` above regulating the accesses to the database `db2` from the outside. Its dynamic version is the following where `policy_flag` is the activation flag:

```
omega2 ← policy_flag, current_usr(U), profile(U,vendor), has_auth(U,db2)
         location(others), accessing(db2).
```

We can also express policies that control which parts of the context an application is allowed to modify, by suitably guarding the facts to be protected by a context policy. For example, the following one bans the users blacklisted from retracting a specific fact F. The first case of `omega3` requires that the fact F always holds in a context where the current user is blacklisted. Any attempt to remove F leads to a violation of the policy, while this is not the case if the user is allowed to retract F.

```
omega3 ← current_usr(U), blacklist(U), F(X).
omega3 ← current_usr(U), ¬ blacklist(U).
```

Suppose that the cryptographic algorithm used by `crypto_query` and `decrypt` are energy consuming. The following policy protects our application to run short of battery, when it wraps lines 15 and 16:

```
psi1 ← battery_level(X), X > threshold.
```

3. ML$_{\text{CoDa}}$

Below, we survey the syntax and the semantics of ML$_{\text{CoDa}}$, along with some small examples to illustrate its peculiar constructs; for more details see [23].

Syntax ML$_{\text{CoDa}}$ consists of two components: 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.

Let Var (ranged over by $x, y, ...$), $Const$ (ranged over by $c, n, ...$) and $Predicate$ (ranged over by $P, ...$) be a set of variables, of basic constants and of predicate symbols, respectively. The syntax is

$$
\begin{array}{rlll}
x \in Var & c \in Const & P \in Predicate & \omega, \psi \in Policies \\
\\
u ::= & x \mid c & u \in Term & \\
A ::= & P(u_1, \ldots, u_n) & A \in Atom & \\
L ::= & A \mid \neg A & L \in Literal & \\
cla ::= & A \leftarrow B. & cla \in Clause & \\
B ::= & \epsilon \mid L, B & B \in ClauseBody & \\
F ::= & A \leftarrow \epsilon & F \in Fact & \\
G ::= & \leftarrow L_1, \ldots, L_n. & G \in Goals &
\end{array}
$$

As usual in Datalog, a term is a variable x or a constant c; an atom A is a predicate symbol P applied to a list of terms; a literal is a positive or a negative atom; a clause is composed by a head, i.e. an atom, and a body, i.e. a possibly empty sequence of literals; a fact is a clause with an empty body and a goal is a clause with empty head. We let goals be ranged over by G, with the intuition that it occurs in a behavioural variation. Context and application policies are ranged over by ω and ψ, respectively. As expected, a policy is obeyed if and only if the corresponding goal holds. For simplicity, we assume that there is a *unique* context policy Ω (referred in the code as `Omega`), resulting from the conjunction of all the relevant policies.

In the following, we assume that each Datalog program is safe and stratified [18] (our world is closed, so we can deal with negation). As anticipated in the introduction, our version of Datalog can express all relational algebras, is fully decidable, and guarantees polynomial response time [27]. Also, many logical languages for defining access control policies follow this stratified-negation-model, e.g. [12,41,26].

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, while $x, f \in Var$ are identifiers for standard variables and functions, with the proviso that $Var \cap DynVar = \emptyset$. The syntax of ML$_{\text{CoDa}}$ is below.

$$
\begin{array}{lll}
Va ::= G.e \mid G.e,\ Va & & \textit{Variations} \\
v \quad ::= & & \textit{Values} \\
 & c \mid \lambda_f x.e \mid & \text{ML-like Values} \\
 & (x)\{Va\} \mid & \text{Behavioural Variations} \\
 & F & \text{Facts}
\end{array}
$$

$e ::=$	*Expressions*
$v \mid x \mid e_1\, e_2 \mid \mathtt{let}\, x\ =\ e_1\, \mathtt{in}\, e_2 \mid \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \mid$	ML-like Expressions
$\tilde{x} \mid$	Parameters
$\mathtt{dlet}\, \tilde{x}\ = e_1\, \mathtt{when}\, G\, \mathtt{in}\, e_2 \mid$	Context-dependent Binding
$\mathtt{tell}(e_1)^l \mid \mathtt{retract}(e_1)^l \mid$	Context update
$e_1 \cup e_2 \mid$	Append Operator
$\#(e_1, e_2) \mid$	Variation Application
$\psi^l[e]$	Policy Framing

To facilitate our static analysis (see Section 5) we associate each **tell**/**retract** and each policy ψ with a different label $l \in Lab_C$; labels do not affect the dynamic semantics, which is defined below.

Standard expressions are evaluated in the usual way. Very briefly, a variable x evaluates to the value it is bound. Both expressions in an application are to be evaluated to values v_1 and v_2, and if $v_1 = \lambda_f x.e$, the evaluation goes on by substituting v_1 for all the (free) occurrences of f in e and v_2 for those of x. The evaluation of a let is that of e_2 where all the (free) occurrences of x have been replaced by the value of e_1. Conditionals evaluate as expected.

The COP-oriented constructs of $\mathrm{ML}_{\mathrm{CoDa}}$ include behavioural variations $(x)\{\,Va\}$, each consisting of a *variation* Va, i.e. a list $G_1.e_1, \ldots, G_n.e_n$ of expressions e_i guarded by Datalog goals G_i (x free in e_i). At run time, the first goal G_i satisfied by the context selects the expression e_i to run (*dispatching*). For instance, in Section 2, we declared the behavioural variation `records`, that returns information on customers, by accessing to different databases depending on the vendor's location.

Context-dependent binding is the mechanism to declare variables whose values depend on the context. The **dlet** construct implements the context-dependent binding of a parameter $\tilde{x}$ to a variation Va. Note that context-dependent binding is designed for expressing adaptability of data, while behavioural variations express adaptability of control flow.

The **tell** and **retract** constructs assert and retract facts in the context, provided that no violation of security occurs.

The append operator $e_1 \cup e_2$ concatenates behavioural variations, so allowing dynamic composition.

A behavioural variation $\#(e_1, e_2)$ applies e_1 to its argument e_2. To do so, the dispatching mechanism is triggered to query the context and to select from e_1 the expression to run, if any.

Furthermore, we use the construct $\psi^l[e]$, called *policy framing*, to guarantee that the context obeys the policy expressed by ψ while running e. With this construct programmers can protect their application from a possible misuse of the running context C. We require ψ to be true in C until e completes its execution, and then the scope of the policy framing is left and the policy de-activated. Also, policy framings can be nested, with the intuition that an expression enclosed in many of them is executed only if the running context obeys them all. The (context) policies ω are instead imposed by the context, to protect its sensible data and devices from a misuse of an application, during its entire evaluation. We will formalise later on how both kinds of policies will be enforced. We presented some examples in the previous section: `omega` is the (part of) context policy that only allows vendors with explicit authorisation to access the database `db2`; `omega3` controls which parts of the context an application is allowed to modify.

Semantics The Datalog component has the standard top-down semantics [18]. Given a context $C \in Context$ and a goal G, we let $C \vDash G\, with\, \theta$ mean that the goal G, under a ground substitution θ, is satisfied in the context C.

(IF1)
$$\frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \to C',\, \texttt{if}\, e_1'\, \texttt{then}\, e_2\, \texttt{else}\, e_3}$$

(IF2)
$$\frac{}{\rho \vdash C,\, \texttt{if}\, true\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \to C,\, e_2}$$

(IF3)
$$\frac{}{\rho \vdash C,\, \texttt{if}\, false\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \to C,\, e_3}$$

(LET1)
$$\frac{\rho \vdash C,\, e_1 \to C',\, e_1'}{\rho \vdash C,\, \texttt{let}\, x\, =\, e_1\, \texttt{in}\, e_2 \to C',\, \texttt{let}\, x\, =\, e_1'\, \texttt{in}\, e_2}$$

(LET2)
$$\frac{}{\rho \vdash C,\, \texttt{let}\, x\, =\, v\, \texttt{in}\, e_2 \to C,\, e_2\{v/x\}}$$

(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. The reduction rules for the ML-like constructs of ML$_{\text{CoDa}}$.

The SOS semantics of ML$_{\text{CoDa}}$ is defined for expressions with no free variables, but possibly with free parameters, thus allowing for openness. To this aim, we have an environment ρ, i.e. a function mapping parameters to variations $DynVar \to Va$. A transition $\rho \vdash C,\, e \to C',\, e'$ represents a single evaluation step. It says that under the environment ρ the expression e is evaluated in the context C and reduces to e' changing C to C'. The initial configuration is $\rho_0 \vdash C,\, e_p$, where ρ_0 contains the bindings for all system parameters, and C results from joining the predicates and facts of the system and of the application e_p.

Figures 4 and 5 show the inductive definitions of the reduction rules for ML$_{\text{CoDa}}$: we briefly comment below on those for our new constructs.

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. Note 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 to which e_2 reduces.

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 follows

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

It inspects a variation from left to right looking for the first goal G satisfied by C, under a substitution θ. If this search succeeds, the dispatching mechanism returns the corresponding expression e and θ. Then,

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

$$\textsc{(Dlet2)}\quad \frac{}{\rho \vdash C,\, \mathtt{dlet}\,\tilde{x} = e_1\,\mathtt{when}\,G\,\mathtt{in}\,v \rightarrow C,\, v} \qquad \textsc{(Par)}\quad \frac{\rho(\tilde{x}) = Va \qquad dsp(C,\ Va) = (e,\, \theta)}{\rho \vdash C,\, \tilde{x} \rightarrow C,\, e\,\theta}$$

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

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

$$\textsc{(Tell1)}\quad \frac{\rho \vdash C,\, e \rightarrow C',\, e'}{\rho \vdash C,\, \mathtt{tell}(e)^l \rightarrow C',\, \mathtt{tell}(e')^l} \qquad \textsc{(Tell2)}\quad \frac{dsp(C \cup \{F\},\, \Omega.()) = ((),\, \emptyset)}{\rho \vdash C,\, \mathtt{tell}(F)^l \rightarrow C \cup \{F\},\, ()}$$

$$\textsc{(Retract1)}\quad \frac{\rho \vdash C,\, e \rightarrow C',\, e'}{\rho \vdash C,\, \mathtt{retract}(e)^l \rightarrow C',\, \mathtt{retract}(e')^l} \qquad \textsc{(Retract2)}\quad \frac{dsp(C \smallsetminus \{F\},\, \Omega.()) = ((),\, \emptyset)}{\rho \vdash C,\, \mathtt{retract}(F)^l \rightarrow C\backslash\{F\},\, ()}$$

$$\textsc{(Append1)}\quad \frac{\rho \vdash C,\, e_1 \rightarrow C',\, e_1'}{\rho \vdash C,\, e_1 \cup e_2 \rightarrow C',\, e_1' \cup e_2} \qquad \textsc{(Append2)}\quad \frac{\rho \vdash C,\, e_2 \rightarrow C',\, e_2'}{\rho \vdash C,\, v \cup e_2 \rightarrow C',\, v \cup e_2'}$$

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

$$\textsc{(Frame1)}\quad \frac{C \vDash \psi \qquad \rho \vdash C,\, e \rightarrow C',\, e' \qquad C' \vDash \psi}{\rho \vdash C,\, \psi^l[e] \rightarrow C',\, \psi^l[e']} \qquad \textsc{(Frame2)}\quad \frac{}{\rho \vdash C,\, \psi^l[v] \rightarrow C,\, v}$$

Figure 5. The reduction rules for the constructs of ML$_{\text{CoDa}}$ concerning adaptation and policies.

$\tilde{x}$ 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.

As an example of context-dependent binding, consider the simple conditional expression $\mathtt{if}\,\tilde{x} = F_2\,\mathtt{then}\,42\,\mathtt{else}\,51$, in an environment ρ that binds the parameter $\tilde{x}$ to $e' = G_1.F_5, G_2.\,F_2$ and in a context C that satisfies the goal G_2, but not G_1:

$$\rho \vdash C,\, \mathtt{if}\,\tilde{x} = F_2\,\mathtt{then}\,42\,\mathtt{else}\,51 \rightarrow C,\, \mathtt{if}\,F_2 = F_2\,\mathtt{then}\,42\,\mathtt{else}\,51 \rightarrow C,\, 42$$

where we first retrieve the binding for $\tilde{x}$ (recall it is e'), with $dsp(C,\, e') = (F_2, \theta)$, for a suitable substitution θ. Since facts are values, we can bind them to parameters and test their equivalence by a conditional expression.

The application of the behavioural variation $\#(e_1, e_2)$ evaluates the subexpressions until e_1 reduces to $(x)\{\, Va\}$ and e_2 to a value v. 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 a example, consider the behavioural variation $(x)\{G_1.c_1, G_2.x\}$ and apply it to the constant c_2 in a context C that satisfies the goal G_2, but not the goal G_1. Since $dsp(C, (x)\{G_1.c_1, G_2.x\}) = (x,\, \theta)$ for some substitution θ, we get:

$$\rho \vdash C,\, \#((x)\{G_1.c_1, G_2.x\}, c_2) \;\rightarrow\; C,\, c_2$$

The rules (TELL1, TELL2) update the context by asserting a fact, that is a value of $\mathrm{ML}_{\mathrm{CoDa}}$; similarly (RETRACT1, RETRACT2) retract a fact (note that labels are immaterial here, and will only be used in the static analysis). The new context C', obtained from C by adding/removing F, is checked against the security policy Ω (recall that we assume to join all the context policies in the single one Ω). In this way we prevent the application from damaging the context, as exemplified below. We can easily reuse our dispatching machinery above: we implement the check as a call to the function dsp where the first argument is C' and the second one is $\Omega.()$, the trivial variation with goal Ω. If this call produces a result, then the evaluation yields the unit value and the new context C'.

The following example shows the reduction of a **retract** in a context $C = \{F_3,\, F_4,\, F_5\}$, under a context policy Ω requiring that the fact F_4 always holds. Let $f = \lambda x.\, \textbf{if}\, e_1\, \textbf{then}\, F_5\, \textbf{else}\, F_4$ and evaluate **retract** $f()^{\,\ell}$. If e_1 evaluates to `false` (without changing the context), the context eventually produced violates the policy in hand, $dsp(C \setminus \{F_4\},\, \Omega.())$ fails, and therefore the evaluation gets stuck:

$$\rho \vdash C,\, \textbf{retract}\, f\,()^{\ell} \rightarrow^{*} C,\, \textbf{retract}\, F_4{}^{\ell} \not\rightarrow$$

where $\not\rightarrow$ means that no transition goes out from C, **retract** F_4^{ℓ}. If, instead, e_1 reduces to `true`, there is no policy violation and the evaluation reduces to `unit`:

$$\rho \vdash C,\, \textbf{retract}\, f\,()^{\ell} \rightarrow^{*} C,\, \textbf{retract}\, F_5^{\ell} \rightarrow \{F_3,\, F_4\}, ()$$

The rules for $e_1 \cup e_2$ sequentially evaluate e_1 and e_2 until they reduce to behavioural variations (rules (APPEND1, 2)). Then, they are concatenated together by renaming bound variables to avoid name captures (rule (APPEND3)). The policy ψ of the application e is also enforced along its evaluation when asserting or retracting a fact by rule (FRAME1). Before performing one of these actions, the ψ has to be shown true in the context C, and the changes made on the context must preserve ψ true, i.e. $C' \vDash \psi$. This check is implemented at run time through a call to the dispatching mechanism, as suggested in the comment to the rules for **tell**/**retract**. The rule (FRAME2) simply discards the framing, so it de-activates the policy ψ as soon as a value is produced.

Back to the example of Section 2, assume that Jane wants to access the database from outside the office, then we have the following computation ($\rightarrow^{+}$ indicates a non empty computation):

$$\rho \vdash C,\, \texttt{\#(records, customers)} \rightarrow^{+} C',\, \texttt{psi}^{4}\texttt{[e]} \rightarrow^{+} C'',\, \texttt{psi}^{4}\texttt{[decrypt(k,data)]}$$

where `e` is the code at lines 15 and 16. From the first to the second configuration the dispatching mechanism queries the context C and selects the second case of `records`. From the third to the fourth configuration the connection to the database is established and the data are retrieved. The framing construct checks the policy `psi` at each step of this computation.

Note that imposing a context policy ω to an application e could be intuitively implemented by wrapping its whole code within the framing $\omega[e]$, at load time. However, monitoring context and application policies is done differently, and separation of concerns and efficiency reasons strongly suggest us keeping them apart.

4. Type and Effect System

We now associate an $\mathrm{ML}_{\mathrm{CoDa}}$ expression with a type, an abstraction called *history expression*, and a function called the *labelling environment*. During the verification phase, the virtual machine uses the history expression to ensure that the dispatching mechanism will always succeed at run time. The labelling environment helps in selecting which portions of the code *may* lead to violations of the security policies, and in instrumenting the code with suitable calls to a run time monitor. First, we briefly present History Expressions and labelling environments, and then the rules of our type and effect system.

4.1. History Expressions

A history expression is a term of a simple process algebra that soundly abstracts program behaviour [63,10]. Here, they over-approximate the sequence of actions that an application may perform over the context at run time, i.e. asserting/retracting facts and asking if a goal holds. History expressions also record the application policies that are to be enforced. We assume that a history expression is uniquely labelled on a given set of Lab_H. Labels will be used to link static actions in histories to the corresponding dynamic actions inside the code; we feel free to omit them when immaterial. The syntax of History Expressions is as follows:

$$\begin{aligned}\mathbb{H} \ni H ::= & \backepsilon \mid \epsilon^l \mid h^l \mid (\mu h.H)^l \mid \mathit{tell}\, F^l \mid \mathit{retract}\, F^l \mid (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \psi^l[H] \mid \Delta \\ \Delta ::= & (\mathit{ask}\, G.H \otimes \Delta)^l \mid \mathit{fail}^l\end{aligned}$$

The empty history expression abstracts programs that 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 ($\backepsilon$ that is unlabelled). With $\mu h.H$ we represent possibly recursive functions, where h is the recursion variable; the "atomic" history expressions $\mathit{tell}\, F$ and $\mathit{retract}\, F$ are for the analogous constructs of $\mathrm{ML}_{\mathrm{CoDa}}$; the non-deterministic sum $H_1 + H_2$ abstracts **`if-then-else`**; the concatenation $H_1 \cdot H_2$ is for sequences of actions that arise, e.g. while evaluating applications; the history expression $\psi^l[H]$ is the abstract version of the security framing; 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 $\mathit{ask}\, G_i$. For instance, the history expression H_{records} intuitively introduced in Section 2 abstracts the behavioural variation `records` in Figure 1.

Given a context C, the behaviour of a history expression H is formalised by the transition system inductively defined in Figure 6. A transition $C, H \to C', H'$ means that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones of [10]: below we only comment on

$$\frac{}{C,\, \epsilon^l \to C,\, \backepsilon} \qquad \frac{}{C,\, tell\, F^l \to C \cup \{F\},\, \backepsilon} \qquad \frac{}{C,\, retract\, F^l \to C \backslash \{F\},\, \backepsilon}$$

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

$$\frac{}{C,\, (\backepsilon \cdot H)^l \to C,\, H} \qquad \frac{C,\, H_1 \to C',\, H_1'}{C,\, (H_1 \cdot H_2)^l \to C',\, (H_1' \cdot H_2)^l} \qquad \frac{}{C, (\mu h.H)^l \to C, H[(\mu h.H)^l/h]}$$

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

$$\frac{C \vDash \psi \qquad C,\, H \to C',\, H' \qquad C' \vDash \psi}{C,\, \psi^l[H] \to C',\, \psi^l[H']} \qquad \frac{}{C,\, \psi^l[\backepsilon] \to C,\, \backepsilon}$$

Figure 6. Semantics of History Expressions

those dealing with the context and with the policy framings. 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 goal is satisfiable, the stuck configuration *C, fail* is reached, meaning that the dispatching mechanism fails.

An action $tell\, F$ reduces to $\backepsilon$ and yields a context C', where the fact F has just been added; similarly for the action $retract\, F$. Differently from what we do in the semantic rules, here we do not consider the possibility of a violation of a context policy ω: a history expression approximates how the application would behave in the absence of any kind of check imposed by the context.

The rules for policy framings are much alike those of the dynamic semantics: a non-empty history expression H can transform into H', provided that in both the starting context C and in the next one C' the policy ψ holds; when $\backepsilon$ is reached, the policy framing is left.

Labelling Environment We assume as given the function $\mu : Lab_H \to \mathbb{H}$ that recovers a construct in a given history expression $H \in \mathbb{H}$ from a label $l \in Lab_H$. Using μ, we can link the occurrences of $tell, retract$ and ψ in a history expression to the corresponding operations in an expression e (labelled on Lab_C, see Section 3), while type-checking e. This correspondence is recorded in the auxiliary labelling environment introduced below. It will later on help in instrumenting the code.

Definition 4.1 (Labelling environment)**.** A *labelling environment* is a (partial) function
$\Lambda : Lab_H \to Lab_C$, defined only if $\mu(l) \in \{tell(F),\, retract(F),\, \psi\}$.
We shall write $\bot$ for the function undefined everywhere.

As anticipated in Section 2, the labelling environment $\{l_1 \mapsto 1, l_2 \mapsto 2, l_3 \mapsto 3, l_4 \mapsto 4, l_5 \mapsto 5\}$ puts in correspondence the labels of the history expression $H_{records}$ with those in the code snippet in Figure 1. Note that labelling environments need not to be injective.

4.2. Typing rules

We assume that each Datalog predicate has a fixed arity and a type (see e.g. [51]). From here onwards, we also 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 variables $x \in G$.

The rules of our type and effect system have:

- the usual environment $\Gamma ::= \emptyset \mid \Gamma, x : \tau$, binding the variables of an expression; $\emptyset$ denotes the empty environment, and $\Gamma, x : \tau$ denotes an environment having a binding for the variable x (x not in the domain of Γ). A standard assumption is to have an initial environment containing the signatures of the APIs.
- a further environment $K ::= \emptyset \mid K, (\tilde{x}, \tau, \Delta, \Lambda)$, that maps a parameter $\tilde{x}$ to a triple consisting of a (i) type, (ii) an abstract variation Δ, used to solve the binding for $\tilde{x}$ at run time, and (iii) a labelling environment Λ that links the $tell, retract$ and ψ occurring in Δ to the corresponding operations in an expression e; $K, (\tilde{x}, \tau, \Delta, \Lambda)$ denotes an environment with a binding for the parameter $\tilde{x}$ ($\tilde{x}$ not in the domain of K).

Our typing judgements have the form

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

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

Our types are either basic $\tau_c \in \{int, bool, unit, \ldots\}$ for constants and variables, or functional types for functions and behavioural variations, or types for facts:

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

Some types are annotated for analysis reasons. In *fact*$_\phi$, the set ϕ contains the facts that an expression can be reduced to at run time (see the semantics rules (TELL2) and (RETRACT2)).

In the type $\tau_1 \xrightarrow{K|H;\,\Lambda} \tau_2$ associated with a function f, the environment K stores the types and the abstract variations of the parameters occurring inside the body of f, and represents a precondition needed to apply it. The history expression H is the latent effect of f, i.e. the sequence of actions that may be performed over the context during the function evaluation. The labelling environment Λ links (some of) the labels of H to those occurring in the body of f.

Similarly, the behavioural variation $bv = (x)\{\,Va\}$ has type $\tau_1 \xRightarrow{K|\Delta;\,\Lambda} \tau_2$, where K is a precondition for applying bv; Δ is an abstract variation, that represents the information used at run time by the dispatching mechanism to apply bv; and Λ is as above.

We now introduce four *partial orderings* $\sqsubseteq_H, \sqsubseteq_\Delta, \sqsubseteq_K, \sqsubseteq_\Lambda$ on H, Δ, K and Λ, respectively, and we often omit the indexes when no ambiguity may arise. Below the symbol $\uplus$ stands for disjoint union, the type ordering $\leq$ is defined in Figure 7, and we assume that *fail* $\otimes\ \Delta = \Delta$, so $\Delta' \otimes \Delta$ will have a single trailing *fail*.

$$H_1 \sqsubseteq_H H_2 \iff \exists H_3 \text{ such that } H_2 = H_1 + H_3$$

(STCONST)
$$\tau_c \leq \tau_c$$

(SFACT)
$$\frac{\phi \subseteq \phi'}{fact_{\phi} \leq fact_{\phi'}}$$

(SFUN)
$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq_K K' \qquad H \sqsubseteq_H H'}{\tau_1 \xrightarrow{K|H} \tau_2 \leq \tau_1' \xrightarrow{K'|H'} \tau_2'}$$

(SVA)
$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad K \sqsubseteq_K K' \qquad \Delta \sqsubseteq_\Delta \Delta'}{\tau_1 \xRightarrow{K|\Delta} \tau_2 \leq \tau_1' \xRightarrow{K'|\Delta'} \tau_2'}$$

Figure 7. The subtyping relation

(TCONST)
$$\frac{}{\Gamma;\, K \vdash c \,:\, \tau_c \triangleright \epsilon;\, \bot}$$

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

(TIF)
$$\frac{\Gamma;\, K \vdash e_1 \,:\, \mathit{bool} \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau \triangleright H_2;\, \Lambda_2 \qquad \Gamma;\, K \vdash e_3 \,:\, \tau \triangleright H_3;\, \Lambda_3}{\Gamma;\, K \vdash \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \,:\, \tau \triangleright H_1 \cdot (H_2 + H_3);\, \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3}$$

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

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

(TAPP)
$$\frac{\begin{array}{c}\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2 \triangleright H_1;\, \Lambda_1 \\ \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2;\, \Lambda_2 \qquad K' \sqsubseteq K\end{array}}{\Gamma;\, K \vdash e_1\, e_2 \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot H_3;\, \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3}$$

Figure 8. Typing rules for ML-like constructs of $\mathrm{ML}_{\mathrm{CoDa}}$.

$$\begin{aligned}
\Delta_1 \sqsubseteq_\Delta \Delta_2 &\iff \exists \Delta_3 \text{ such that } \Delta_2 = \Delta_1 \otimes \Delta_3 \\
K_1 \sqsubseteq_K K_2 &\iff (\,(\tilde{x},\, \tau_1,\, \Delta_1,\, \Lambda_1) \in K_1 \implies \\
&\qquad (\tilde{x},\, \tau_2,\, \Delta_2,\, \Lambda_2) \in K_2 \text{ and } \tau_1 \leq \tau_2 \text{ and } \Delta_1 \sqsubseteq_\Delta \Delta_2 \text{ and } \Lambda_1 \sqsubseteq_\Lambda \Lambda_2\,) \\
\Lambda_1 \sqsubseteq_\Lambda \Lambda_2 &\iff \exists \Lambda_3 \text{ such that } \mathit{dom}(\Lambda_3) \cap \mathit{dom}(\Lambda_1) = \emptyset \text{ and } \Lambda_2 = \Lambda_1 \uplus \Lambda_3
\end{aligned}$$

The rules of our type and effect system inherited from ML are displayed in Figure 8, while the others are introduced in Figures 9 and 10.

ML-inherited rules We briefly comment on the most relevant rules in Figure 8. In the rules (TCONST), (TVAR) and (TABS) the labelling environment is empty. The rule (TIF) is standard, apart from the resulting labelling environment that is the (disjoint) union of those of the components e_i. Same for (TLET) and for

(TFACT)

$$\frac{}{\Gamma;\, K \vdash F \;:\; \mathit{fact}_{\{F\}} \triangleright \epsilon;\, \bot}$$

(TTELL)

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

(TRETRACT)

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

(TVARIATION)

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

Figure 9. Typing rules for context updates

(TAPP) where the labelling environment of the latent effect is also considered. Note that in the conclusion of (TAPP), the precondition of the latent effect of e must be included in the environment K to guarantee that all parameters will be duly bounded. The rule (TABS) is standard, except for the way the environments K and Λ are handled. As usual, the actual effect guessed in the premise becomes latent in the conclusion of the rule, while the effect and the labelling environment become empty; note that in the premise the guessed parameter environment is the one used for typing e, while the guessed effect H and the guessed labelling environment Λ are those of e.

Context rules The rules for the constructs that handle the context are in Figure 9. A few comments follow. The rule (TFACT) gives a fact F type *fact* annotated with $\{F\}$ and the empty effect. The rule (TTELL) asserts that the expression $\texttt{tell}(e)$ has type *unit*, provided that the type of e is fact_{ϕ}. The overall effect is obtained by combining the effect of e with the non deterministic summation of $\texttt{tell}\, F$, where F is any of the facts in the type of e. The current labelling environment is extended with links from the elements of ϕ to the label of $\texttt{tell}(e)$. Similarly for (TRETRACT). 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 variables z of G_i). Note that all subexpressions e_i have the same type τ_2, but of course they have different effects and labelling environments. We also require that the abstract variation Δ results from concatenating $\mathit{ask}\, G_i$ with the effect computed for e_i. The arrow in the type of the behavioural variation is annotated by K', Δ and by the union of all the labelling environments Λ_i. 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 $\Gamma, x : \mathit{int}$

(goals have no variables) and the guessed environment K'. Hence, the type of bv_1 will be $int \xRightarrow{K'|\Delta} \tau$ with $\Delta = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$, while the effect will be empty.

Rules for adaptation and security In Figure 10 we list the rules dealing with adaptation and security. The rule (TSUB) allows us to freely enlarge types and effects by applying the subtyping and subeffecting rules (and the orderings on them). Also the labelling environment can be enlarged, and it is required to be defined on all the labels of the (larger) history expression H. The rule (TDVAR) associates $\tilde{x}$ with the triple stored 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 as e_2, while the labelling environment is the union of those of e_1, e_2 and of the one coming from $\tilde{x}$. 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. 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 with type τ_1. We get a type if the environment K', which acts as a precondition, is included in K according to $\sqsubseteq$. The type of the behavioural variation application is that 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. The overall labelling environment results from joining those of the subexpressions and of the latent effect.

(TSUB)

$$\frac{\Gamma;\, K \vdash e \,:\, \tau' \triangleright H';\, \Lambda' \qquad \tau' \leq \tau \qquad H' \sqsubseteq H \qquad \Lambda' \sqsubseteq \Lambda}{\Gamma;\, K \vdash e \,:\, \tau \triangleright H;\, \Lambda} \quad lab(H) \subseteq dom(\Lambda)$$

(TDVAR)

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

(TDLET)

$$\frac{\Gamma, \overrightarrow{y} : \overrightarrow{\tau};\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K, (\tilde{x},\, \tau_1,\, \Delta',\, \Lambda') \vdash e_2 \,:\, \tau \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash \mathtt{dlet}\, \tilde{x} \,=\, e_1\, \mathtt{when}\, G\, \mathtt{in}\, e_2 \,:\, \tau \triangleright H_2;\, \Lambda_2}$$

where $\gamma(G) = \overrightarrow{y} : \overrightarrow{\tau}$
$$(\Delta', \Lambda') = \begin{cases} (G.H_1 \otimes \Delta, \Lambda \uplus \Lambda_1) & \text{if } K(\tilde{x}) = (\tau_1,\, \Delta,\, \Lambda) \\ (G.H_1 \otimes fail,\, \Lambda_1) & \text{if } \tilde{x} \notin K \end{cases}$$

(TAPPEND)

$$\frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xRightarrow{K'|\Delta_1;\Lambda_3} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \xRightarrow{K'|\Delta_2;\Lambda_4} \tau_2 \triangleright H_2;\, \Lambda_2}{\Gamma;\, K \vdash e_1 \cup e_2 \,:\, \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2;\Lambda_3 \uplus \Lambda_4} \tau_2 \triangleright H_1 \cdot H_2; \Lambda_1 \uplus \Lambda_2}$$

(TVAPP)

$$\frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xRightarrow{K'|\Delta;\Lambda_3} \tau_2 \triangleright H_1;\, \Lambda_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2;\, \Lambda_2 \qquad K' \sqsubseteq K}{\Gamma;\, K \vdash \#(e_1, e_2) \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot \Delta;\, \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3}$$

(TFRAME)

$$\frac{\Gamma;\, K \vdash e :\, \tau \triangleright H;\, \Lambda}{\Gamma;\, K \vdash \psi^{l}[e] :\, \tau \triangleright \psi^{l'}[H];\, \Lambda \uplus [l' \mapsto l]}$$

Figure 10. Typing rules for adaptation and security

Finally, in the single rule for the security framing (TFRAME), the effect computed for e is wrapped with the policy ψ, and the labelling environment is the union of that of e with a link from the occurrence of ψ in the history expression to the corresponding one in the code.

4.3. Soundness

Our type and effect system is sound with respect to the operational semantics of $\mathrm{ML}_{\mathrm{CoDa}}$. To prove this result, we first introduce a notion for typing a dynamic environment ρ in a way consistent with the type environments Γ and K.

Definition 4.2 (Typing dynamic environment)**.** Given the type environments Γ and K, we say that the dynamic environment ρ has type K under Γ (in symbols $\Gamma \vdash \rho \ : \ K$) if and only if

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

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

Theorem 4.1 (Preservation)**.**
Let e be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e and $\Gamma \vdash \rho \ : \ K$.
If $\Gamma;\ K \vdash e_s \ : \ \tau \triangleright H_s; \Lambda_s$ and $\rho \vdash C,\ e_s \to C',\ e_s'$, then $\Gamma;\ K \vdash e_s' \ : \ \tau \triangleright H_s'; \Lambda_s'$, $C, H_s \to^* C',\ H''$ for some $H'' \sqsubseteq H_s'$ and $\Lambda_s' \sqsubseteq \Lambda_s$.

The next corollary ensures that the effect computed for e soundly approximates the actions that may be performed over the context during the evaluation of e. Also, the type of the expression obtained after some evaluation steps is the same of e, because of Theorem 4.1, and of course the obtained label environment is included in Λ_s.

Corollary 1 (Over-approximation)**.** *Let e be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e and $\Gamma \vdash \rho \ : \ K$.*
If $\Gamma; K \vdash e \ : \ \tau \triangleright H; \Lambda$ and $\rho \vdash C,\ e \to^* C',\ e'$, then $\Gamma; K \vdash e' \ : \ \tau \triangleright H'; \Lambda'$ and there exists a sequence of transitions $C,\ H \to^* C',\ H''$, for some $H'' \sqsubseteq H'$ and $\Lambda' \sqsubseteq \Lambda$.

The Progress Theorem stated below assumes that the effect H never evaluates to *fail*, namely it is *viable*. We will see that this implies that while executing the expression e that has effect H, the dispatching mechanism will always succeed. The control flow analysis of Section 5 will check viability of history expressions. To establish our result, we also require that no policy is violated along evaluation. We shall guarantee that this is the case through the analysis presented in Section 5, possibly resorting to a reference monitor; see also Property 6.1.

From now onwards, we shall use the following abbreviations. Given an environment ρ, a context C and a closed expression e, we write

- $\rho \vdash C,\ e \not\to$ if and only if there exists no C', e' such that $\rho \vdash C,\ e \to C',\ e'$;
- e *violates no policies* if and only if the rule (FRAME1) successfully applies to all its derivatives e'.

Theorem 4.2 (Progress)**.**
Let e be a closed expression such that $\Gamma; K \vdash e \ : \ \tau \triangleright H; \Lambda$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e, and $\Gamma \vdash \rho \ : \ K$.
If $\rho \vdash C,\ e \not\to$; e violates no policies; and H is viable for C (i.e. $C,\ H \not\to^+ C'$, fail), then e is a value.

We are now ready to state the theorem that ensures the semantic correctness of our approach.

Theorem 4.3 (Correctness).
Let e_s be a closed expression such that $\Gamma; K \vdash e_s \, : \, \tau \triangleright H_s; \Lambda_s$; let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and that $\Gamma \vdash \rho \, : \, K$; finally let C be a context such that $C, H_s \not\rightarrow^+ C', fail$. Then either the computation of e_s terminates yielding a value ($\rho \vdash C, e_s \rightarrow^* C'', v$) or it diverges, but it never gets stuck.

5. Load time Analysis

Our execution model for $\mathrm{ML_{CoDa}}$ extends the one in [23]: the compiler [1] produces a quadruple $(C,\, e,\, H,\, \Lambda)$ given by (i) the application context C; (ii) the object code e; (iii) the history expression H over-approximating the behaviour of e; and (iv) the labelling environment Λ associating labels of H with those in the code. Given this quadruple, the virtual machine performs the following two steps at load time:

- *linking*: to resolve system variables and constructs the initial context $\overline{C}$ (combining C and the system context); and
- *verification*: to build, from H, a graph $\mathcal{G}$ that describes the possible evolutions of $\overline{C}$. This graph will be used to support a further analysis, and code instrumentation. Instrumenting the code will allow us to call *on need* a reference monitor that prevents actions violating the security policies in force.

Technically, we compute $\mathcal{G}$ through a static analysis, specified in terms of Flow Logic [54]. To ease the formal development, we assume below that all the bound variables occurring in a history expression are distinct. So it is straightforward to define, by structural induction, a function $\mathbb{K}$ mapping a variable h^l to the history expression $(\mu h.H_1^{l_1})^{l_2}$ that introduces it.

The static approximation is represented by a quadruple $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$, called *estimate* for H, with

$$\Sigma_\circ, \Sigma_\bullet \colon Lab_H \to \wp(Context \cup \{\circledast\}) \qquad \text{and} \qquad \Psi_\circ, \Psi_\bullet \colon Lab_H \to \wp(Policies)$$

where $\circledast$ is the distinguished "failure" context representing a dispatching failure. For each label l we define the following four sets:

- the *pre-ctx* $\Sigma_\circ(l)$ and the *post-ctx* $\Sigma_\bullet(l)$, containing the contexts possibly arising *before* and *after* evaluating H^l, respectively;
- the *pre-pol* $\Psi_\circ(l)$ and the *post-pol* $\Psi_\bullet(l)$, containing the application policies possibly active *before* and *after* evaluating H^l, respectively.

The analysis is specified by a set of clauses upon judgements of the following form

$$(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H^l$$

with the intended meaning that the four functions $\Sigma_\circ, \Sigma_\bullet, \Psi_\circ$ and $\Psi_\bullet$ form an acceptable analysis estimate for the history expression H^l. This is the first step for proving an estimate to soundly approximate the behaviour of H as it will be formalised below through the notion of validity.

[1] Our prototype [15] does not fully integrate the F# type system with ours; presently we only have an early implementation of our analyses.

(AEPS)

$$\frac{\begin{array}{c}\Sigma_{\circ}(l) \subseteq \Sigma_{\bullet}(l) \\ \Psi_{\circ}(l) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \epsilon^{l}}$$

(ASEQ1)

$$\frac{\begin{array}{ccc} \multicolumn{3}{c}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H_1^{l_1} \qquad (\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H_2^{l_2}} \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_1) & \Sigma_{\bullet}(l_1) \subseteq \Sigma_{\circ}(l_2) & \Sigma_{\bullet}(l_2) \subseteq \Sigma_{\bullet}(l) \\ \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_1) & \Psi_{\bullet}(l_1) \subseteq \Psi_{\circ}(l_2) & \Psi_{\bullet}(l_2) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models (H_1^{l_1} \cdot H_2^{l_2})^{l}}$$

(ASEQ2)

$$\frac{\begin{array}{cccc} \multicolumn{4}{c}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H_2^{l_2}} \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_2) & \Sigma_{\bullet}(l_2) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_2) & \Psi_{\bullet}(l_2) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models (\backepsilon \cdot H_2^{l_2})^{l}}$$

(ASUM)

$$\frac{\begin{array}{cccc} \multicolumn{4}{c}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H_1^{l_1} \qquad (\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H_2^{l_2}} \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_1) & \Sigma_{\bullet}(l_1) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_1) & \Psi_{\bullet}(l_1) \subseteq \Psi_{\bullet}(l) \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_2) & \Sigma_{\bullet}(l_2) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_2) & \Psi_{\bullet}(l_2) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models (H_1^{l_1} + H_2^{l_2})^{l}}$$

(AREC)

$$\frac{\begin{array}{cccc} \multicolumn{4}{c}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H^{l_1}} \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_1) & \Sigma_{\bullet}(l_1) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_1) & \Psi_{\bullet}(l_1) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models (\mu h.H^{l_1})^{l}}$$

(AVAR)

$$\frac{\mathbb{K}(h) = (\mu h.H)^{l'} \qquad \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l') \qquad \Sigma_{\bullet}(l') \subseteq \Sigma_{\bullet}(l) \qquad \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l') \qquad \Psi_{\bullet}(l') \subseteq \Psi_{\bullet}(l)}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models h^{l}}$$

Figure 11. Specification of the analysis for History Expressions associated with standard ML expressions.

The notion of acceptability has been used in [22,23] to check whether the history expression H, hence the expression e it is an abstraction of, will never fail in a given initial context C (made of the system and the application contexts) because the dispatching mechanism does not succeed. Here, we also detect which actions, if any, may violate the security policies in force.

In the following we introduce into two parts (shown in Figures 11 and 12, respectively) the set of inference rules that validate the correctness of a given estimate $\mathcal{E} = (\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet})$. As expected, the checks in the clauses mimic the semantic evolution of the history expression in a given context, by modelling the semantic preconditions and the consequences of the possible reductions.

Rules for the standard construct The rules for the history expressions that abstract standard ML expressions are in Figure 11. The rule (AEPS) says that the estimate $\mathcal{E}$ is acceptable for the "syntactic" ϵ^{l} if the pre-sets are included in the corresponding post-sets.

The rules (ASEQ1) and (ASEQ2) handle the sequential composition of history expressions. The first rule states that $(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet})$ 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-ctx of H_1 must include the pre-ctx of H and the pre-ctx of H_2 includes the post-ctx of H_1; finally the post-ctx of H includes that of H_2. Just the same condition is required for the pre- and

(ANIL)

$$\frac{}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \backepsilon}$$

(ATELL)

$$\frac{\begin{array}{c}\forall C \in \Sigma_{\circ}(l) \\ C \cup \{F\} \in \Sigma_{\bullet}(l) \qquad \Psi_{\circ}(l) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \mathit{tell}\, F^{l}}$$

(ARETRACT)

$$\frac{\begin{array}{c}\forall C \in \Sigma_{\circ}(l) \\ C \backslash \{F\} \in \Sigma_{\bullet}(l) \qquad \Psi_{\circ}(l) \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \mathit{retract}\, F^{l}}$$

(AASK1)

$$\frac{\begin{array}{ccccc} & & \forall C \in \Sigma_{\circ}(l) & & \\ & (C \models G \implies (\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H^{l_1} & & & \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_1) & \Sigma_{\bullet}(l_1) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_1) & \Psi_{\bullet}(l_1) \subseteq \Psi_{\bullet}(l)\ ) & \\ & (C \not\models G \implies (\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \Delta^{l_2} & & & \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l_2) & \Sigma_{\bullet}(l_2) \subseteq \Sigma_{\bullet}(l) & \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l_2) & \Psi_{\bullet}(l_2) \subseteq \Psi_{\bullet}(l)\ ) & \end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models (askG.H^{l_1} \otimes \Delta^{l_2})^{l}}$$

(AASK2)

$$\frac{\divideontimes \in \Sigma_{\bullet}(l) \qquad \Psi_{\circ}(l) \subseteq \Psi_{\bullet}(l)}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \mathit{fail}^{l}}$$

(AFRAME)

$$\frac{\begin{array}{c}(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models H^{l'} \\ \Sigma_{\circ}(l) \subseteq \Sigma_{\circ}(l') \qquad \Sigma_{\bullet}(l') \subseteq \Sigma_{\bullet}(l) \qquad \Psi_{\circ}(l) \subseteq \Psi_{\circ}(l') \qquad \{\psi\} \subseteq \Psi_{\circ}(l) \qquad \Psi_{\bullet}(l') \setminus \{\psi\} \subseteq \Psi_{\bullet}(l)\end{array}}{(\Sigma_{\circ}, \Sigma_{\bullet}, \Psi_{\circ}, \Psi_{\bullet}) \models \psi^{l}[H^{l'}]}$$

Figure 12. Specification of the analysis for History Expressions for adaptation and security.

post-pol. The second rule states that $\mathcal{E}$ is acceptable for $H = (\backepsilon \cdot H_1^{l_2})^{l}$ if it is acceptable for H_1 and the pre-ctx (pre-pol, respectively) of H_1 includes that of H, while the post-ctx (post-pol, resp.) 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 each H_1 and H_2; the pre-ctx of H is included in the pre-sets of H_1 and H_2; the post-ctx of H includes those of H_1 and H_2; and the same inclusions hold between the pre- and the post-pol.

By the rule (AREC), $\mathcal{E}$ is acceptable for $H = (\mu h.H_1^{l_1})^{l}$ if it is valid for $H_1^{l_1}$, the pre-ctx (pre-pol, resp.) of H_1 includes that of H; and the post-ctx (post-pol, resp.) of H includes that of H_1.

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

Rules for adaptation and security We now introduce and discuss the rules for adaptation and security, displayed in Figure 12.

The rule (ANIL) says that every pair of functions is an acceptable estimate for the "semantic" empty history expression $\backepsilon$ (the empty history expression reached after a successful execution).

In the rule (ATELL), we check whether the context C is in the pre-ctx, and $C \cup \{F\}$ is in the post-set, while the pre-pol have to be included in the post-pol; similarly for(ARETRACT), where $C \backslash \{F\}$ should be in the post-set.

The rules (AASK1) and (AASK2) handle the abstract dispatching mechanism. The first states that $\mathcal{E}$ is acceptable for $H = (askG.H_1^{l_1} \otimes \Delta^{l_2})^l$, provided that, for all C in the pre-ctx of H, if the goal G succeeds in C then the pre-ctx of H_1 includes that of H and the post-ctx of H includes that of H_1. Otherwise, the pre-ctx of Δ^{l_2} must include the pre-ctx of H and the post-ctx of Δ^{l_2} is included in that of H. The rule (AASK2) requires $\divideontimes$ to be in the post-ctx of $fail^l$. Note that this implies that the dispatching mechanism may fail at run time. Needless to say, in both rules the same inclusions must hold between the pre- and the post-pol.

Finally, through the rule (AFRAME) we say that $\mathcal{E}$ is acceptable for $H' = \psi^l[H^{l'}]$ if it is such for $H^{l'}$; if the pre-ctx (pre-pol, resp.) of H' is included in the pre-ctx (pre-pol, resp.) of $H^{l'}$; if the post-ctx of H' includes that of $H^{l'}$; if additionally the policy ψ belongs to the pre-pol of H' and the post-pol of H' *without* ψ (recall that the framing is left) is included in the post-pol of $H^{l'}$. In other words, all the policies that have to be enforced before entering the policy framing for ψ are still active when the framing is left, and furthermore ψ is in force entering the framing and is de-activated when leaving it.

Semantic properties We now formalise the notion of valid estimate for a history expression, plugged in a context; we prove that there always exists a minimal valid analysis estimate; and that a valid estimate is correct with respect to the operational semantics of history expressions. Valid estimates then soundly approximate the behaviour of history expressions.

Definition 5.1 (Valid analysis estimate). Given H^l and an initial context C, we say that the quadruple $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ is a *valid analysis estimate for H and C* iff $C \in \Sigma_\circ(l)$ and $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H^l$.

It is easy to partially order the set of analysis estimates by set inclusion, component-wise, and to prove that they are a Moore family. Since such families contain a least element (as well as a greatest one), there always exists a least choice of $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ that is acceptable. The existence of a least estimate is stated below.

Theorem 5.1 (Existence of estimates).
Given H^l and an initial context C, the set $\{\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \mid \mathcal{E} \models H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.

The next theorem guarantees that the analysis estimates are preserved under execution steps.

Theorem 5.2 (Subject Reduction).
Let H_1^l be a closed history expression and let $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_1^l$. If for all $C \in \Sigma_\circ(l)$ we have that $C, H_1^l \to C', H_2^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_2^{l'}$; $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$; $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$; $\Psi_\circ(l) \subseteq \Psi_\circ(l')$; and $\Psi_\bullet(l') \subseteq \Psi_\bullet(l)$.

Viability of history expressions We now define when a history expression H is viable for an initial context C, i.e. when it passes the verification phase at load time, following [22,23]. Actually, a viable H never evaluates to *fail*, and the expression the behaviour of which it abstracts will never get stuck, because the dispatching mechanism fails.

Definition 5.2 (Viability). Let H be a history expression; let *lfail*(H) be the set of labels of the *fail* subterms in H; and let C be an initial context. We say that H is *viable* for C if the minimal valid analysis estimate $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ is such that $\divideontimes \in \Sigma_\bullet(l)$ for no $l \in dom(\Sigma_\bullet) \backslash lfail(H)$.

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
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	$\emptyset$	$\emptyset$
8	$\emptyset$	$\emptyset$
9	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_5, F_8\},\{F_2, F_5\}\}$

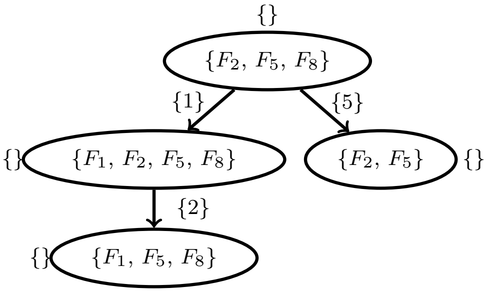

Figure 13. The analysis estimate ($\Psi_\circ(l)$ and $\Psi_\bullet(l)$ are always $\emptyset$ and are omitted) and the evolution graph $\mathcal{G}_p$ for the history expression $H_p = ((tell\, F_1^1 \cdot retract\, F_2^2)^3 + (ask\, F_5.retract\, F_8^5 \otimes ask\, F_3.retract\, F_4^6 \otimes fail^7)^4)^8$ and for the initial context $C = \{F_2,\, F_5,\, F_8\}$.

Note that if the minimal valid estimate satisfies the requirement above, then no other valid estimate will violate it, because of the Moore family property.

Property 5.3. *Let H be a history expression associated with an application e. If H be viable for a context C, then it is never the case that $C,\ H \to^* fail$.*

Example 5.1. To illustrate how viability is checked, consider the following history expression and the initial context $C = \{F_2,\, F_5,\, F_8\}$, only consisting of facts:

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

The left part of Figure 13 shows the values of $\Sigma^1_\circ(l)$ and $\Sigma^1_\bullet(l)$ for H_p, while those for $\Psi_\circ(l)$ and for $\Psi_\bullet(l)$ are omitted being the emptyset everywhere. Note that the pre-ctx of $tell\, F_1^1$ includes $\{F_2,\, F_5,\, F_8\}$, and the post-ctx also includes a set containing F_1. Also, the pre-ctx of $retract\, F_8^5$ includes $\{F_2,\, F_5,\, F_8\}$, while the post-ctx includes $\{F_2,\, F_5\}$. The column for $\Sigma_\bullet$ contains $\divideontimes$ nowhere, so H_p is viable for C.

Instead, an example of a history expression that fails to pass the verification phase when put in the same initial context $C = \{F_2,\, F_5,\, F_8\}$ is

$$H'_p = (\,\psi_0^8[\,\psi_1^9[tell\, F_1^1] \cdot retract\, F_2^2)^3] + (ask\, F_3.retract\, F_4^5 \otimes fail^6)^4)^7$$

where there are two trivial policies: ψ_0 respected if and only if F_2 holds, and ψ_1 if and only if F_5. A functional failure occurs because the fact F_3 holds in no reachable context. This is reflected by the occurrences of $\divideontimes$ in $\Sigma^2_\bullet(4)$ and $\Sigma^2_\bullet(6)$, and in $\Sigma^2_\bullet(7)$, as shown in the upper part of Figure 14. The functional failure may occur because the goal F_3 does not hold in C, and indeed H'_p is not viable. This phase also detects that a policy violation may occur, and if the program had passed the verification phase, at run time the reference monitor would have forbidden the action corresponding to $retract\, F_2^2$ to occur, because it violates ψ_0.

We now exploit the valid estimates computed by the above analysis to build up the evolution graph $\mathcal{G}$ for a history expression H in an initial context C. It describes how C may be updated at run time, paving the way to security enforcement. In the following definition we use the function μ introduced right before in Definition 4.1 that recovers a construct in H from a label l.

	$\Sigma_\circ^2$	$\Sigma_\bullet^2$	$\Psi_\circ^2$	$\Psi_\bullet^2$
1	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_2,\, F_5,\, F_8\}\}$	$\{\psi_0, \psi_1\}$	$\{\psi_0, \psi_1\}$
2	$\{\{F_1,\, F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}\}$	$\{\psi_0\}$	$\{\psi_0\}$
3	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}\}$	$\{\psi_0\}$	$\{\psi_0\}$
4	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{⋇\}$	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
6	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{⋇\}$	$\emptyset$	$\emptyset$
7	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}, ⋇\}$	$\emptyset$	$\emptyset$
8	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_5,\, F_8\}\}$	$\{\psi_0\}$	$\emptyset$
9	$\{\{F_2,\, F_5,\, F_8\}\}$	$\{\{F_1,\, F_2,\, F_5,\, F_8\}\}$	$\{\psi_0, \psi_1\}$	$\{\psi_0\}$

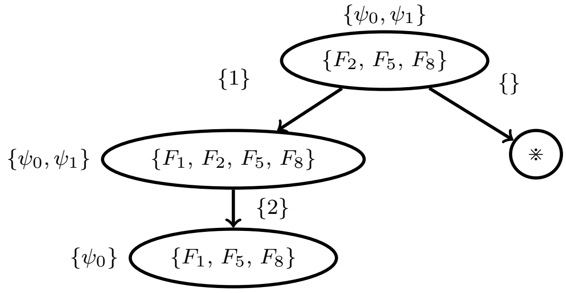

Figure 14. The analysis estimate and the evolution graph $\mathcal{G}'_p$ for the initial context $C = \{F_2,\, F_5,\, F_8\}$, and for the history expression $(\,\psi_0^8[\,\psi_1^9[tell\,F_1^1]\cdot retract\,F_2^2)^3] + (ask\,F_3.retract\,F_4^5 \otimes fail^6)^4)^7$.

Definition 5.3 (Evolution Graph)**.** Let C be a context and H^l be a history expression; let $Fact^*$ and Lab_H^* be the sets of facts and of labels occurring in H, respectively; and let $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ be a valid analysis estimate for H and C.
The *evolution graph of C under H* is $\mathcal{G} = (N, E, L_\Psi, L_E)$, with nodes in N labelled by L_Ψ and edges in E labelled by L_E, where

$$\begin{array}{lll}
N & = & \bigcup_{l \in Lab_H^*} (\Sigma_\circ(l) \cup \Sigma_\bullet(l)) \\
E & = & \{(C_1, C_2) \mid \exists F \in Fact^*,\ l \in Lab_H^* \text{ such that } \mu(l) \in \{tell(F),\ retract(F)\} \wedge \\
 & & C_1 \in \Sigma_\circ(l) \wedge (C_2 \in \Sigma_\bullet(l) \vee C_2 = ⋇)\} \\
L_\Psi : & & N \to \mathcal{P}(Policies) \\
 & & \Psi_\circ(l) \cup \Psi_\bullet(l) \subseteq L_\Psi(n) \text{ iff } n \in \Sigma_\circ(l) \cup \Sigma_\bullet(l) \\
L_E : & & E \to \mathcal{P}(Labels) \\
 & & l \in L_E(t)\ \mathit{iff}\ t = (C_1,\, C_2) \in E,\ \wedge C_1 \in \Sigma_\circ(l) \wedge \mu(l) \neq\ fail
\end{array}$$

Intuitively, the nodes of $\mathcal{G}$ are sets of contexts reachable from C by running H, and we label them with the policies of the application that are to be enforced. An edge between two nodes C_1 and C_2 says that C_2 is obtained from C_1, through an action $tell$ or $retract$, a reference to which is recorded in the edge label. Note in passing that the graph is a compact and easy way to manipulate representation of the information computed by the analysis, and that it paves the ways for more involved verification mechanisms. For example, one could check CTL formulas on it, where the atomic propositions are Datalog goals. Consider again the history expressions H_p and H'_p and their evolution graphs $\mathcal{G}_p$ and $\mathcal{G}'{}_p$ in Figures 13 and 14. In

$\mathcal{G}_p$, from the initial context C there is an edge labelled $\{1\}$ to $C \cup \{F_1\}$, because of $tell\, F_1^1$, and there is an edge labelled $\{5\}$ to the $C \setminus F_8$, because of $retract\, F_8^5$. Of course, all the nodes are labelled by $\emptyset$, since no policy occurs in H_p. A simple reachability algorithm suffices in showing H_p viable for C (even less, as $\circledast$ does no occur in $\mathcal{G}_p$). Instead, $\circledast$ is a reachable node of $\mathcal{G}'_p$ showing H'_p not viable. In addition, in H'_p there are two application policies, and so the nodes of the graph $\mathcal{G}'_p$ are labelled with those that must be checked there, in particular $\{F_1,\, F_2,\, F_5,\, F_8\}$ is labelled by $\{\psi_0,\, \psi_1\}$. It is now equally simple showing that the action $retract\, F_2^2$ has to be blocked, since $\{F_1,\, F_5,\, F_8\} \not\vDash \psi_0$.

Note that the labels on the edges of a graph $\mathcal{G}$ indicate a super-set of actions $tell$ or $retract$ that may lead to a context violating the required policies, while those on the nodes are a super-set of the application policies in force. As a consequence, their correspondence with the labels in the code can be exploited at load time to compute which security checks are necessary. In the next section, we will present a run time reference monitor, defined *within* $\mathrm{ML}_{\mathrm{CoDa}}$, that is only switched on need.

6. Code instrumentation

We use the information stored in the labels of the evolution graph $\mathcal{G}$ for a history expression H of an application e and a context C, to anticipate at load time (a super-set of) the security checks needed when e will run in C. We suggest instrumenting the original code *before deploying it* with checks that will only be activated if the policy actually needs to be enforced, be it a context or an application policy. At load time, we use the labels of the arcs of $\mathcal{G}$ to locate the abstract $tell$ or $retract$ actions in H that may lead to a context violating a given context policy ω. Similarly, we check every application of a policy ψ in the label of a node n against each context included in n to single out those that may violate ψ. As we are statically analysing the behaviour of the approximation H, we need to recover in the code e the actual actions that may lead to these violations, and for that we will resort to the component Λ of the provided valid estimate for H and C.

Assume then that $\{\omega_i\}$ are the context policies and that $\{\psi_i\}$ are the application policies occurring within e, the application in hand. Also, let H be the history expression and Λ be labelling environment resulting from having type checked e. As seen in the previous section, a node n of a graph $\mathcal{G}$ represents a set of contexts $\{C_i\}$ reachable while executing e.

We first *statically* verify whether each context in n satisfies ω. If this is not the case, we consider as risky all the edges with target n and the set R_Ω of their labels — clearly over-approximating bad behaviours. We then consider the application policies active in n, i.e. $L_\Psi(n)$, and we check whether they are obeyed by the contexts represented by n. The set of the application policies violated is called R_Ψ. Now, the labelling environment Λ determines those actions in the code that require monitoring during the execution. Formally, given the graph $\mathcal{G} = (N, E, L_\Psi, L_E)$ for H and C, we let

$$R_\Omega(n) = \{(l, \omega) \mid (n, n') \in E \wedge l = \Lambda(L_E(n, n')) \wedge n' \not\vDash \omega\}$$
$$R_\Psi(n) = \{(l, \psi) \mid (n, n') \in E \wedge l = \Lambda(L_E(n, n')) \wedge \psi \in L_\Psi(n)) \wedge (n \not\vDash \psi \vee n' \not\vDash \psi)\}$$

The property below directly follows from Property 5.3 and provides us with the conditions for an application to never fail because of unresolved behavioural variations and to never violate the required policies.

Property 6.1. *Given an expression e with associated history expression H, let $\mathcal{G} = (N, E, L_\Psi, L_E)$ be the evolution graph of H and C.*
If H is viable and for all nodes n the set $R_\Omega(n) \cup R_\Psi(n) = \emptyset$, then the evaluation of e never gets stuck.

Consider again the history expressions of Example 5.1 and assume that there is a context policy ω that holds if and only if $F_1 \vee F_2$. For all $n \in \mathcal{G}$, we have that $R_\Omega(n) = \emptyset$, signalling that the reference monitor for ω can safely be switched off, while running the actual code of which both history expressions are an abstraction.

Now consider H'_p. We have $R_\Psi(\{F_1, F_2, F_5, F_8\}) = R_\Psi(\{F_1, F_5, F_8\}) = (2, \psi_1)$, while for all other nodes the value is $\emptyset$. Also here we see that the application policy ψ_0 will always be obeyed, so the reference monitor for it can be kept off. Instead, the execution of the action identified through $\Lambda(2)$ requires checking the policy ψ_1 that is actually violated.

We now explain our lightweight form of code instrumentation, which is not standard, because it does not operate on the object code. As said in Section 2, the compiler labels the source code e and generates specific calls to *monitoring* procedures, so implementing the reference monitor. We remark that ML$_{\text{CoDa}}$ requires no further mechanism to do that, as it will be clear shortly.

We define a procedure, called `check_violation(e,l)`, for verifying if a policy is satisfied. It takes an expression **`tell`**/**`retract`** and its label `l` as parameters (and returns the type $unit$, and empty effect and labelling environment). At load time, we assign two global masks $\texttt{risky}_\Omega[\texttt{l}]$ and $\texttt{risky}_\Psi[\texttt{l}]$ to each label `l` of a **`tell`**/**`retract`** action occurring in the source code. The values of these masks contain the set of the context and the application policies, respectively, that are worth checking. Their definition is as follows:

$$\texttt{risky}_\Omega[\texttt{l}] = \{\omega \mid (l, \omega) \in \bigcup_{n \in N} R_\Omega(n)\} \qquad \texttt{risky}_\Psi[\texttt{l}] = \{\psi \mid (l, \psi) \in \bigcup_{n \in N} R_\Psi(n)\}$$

Note that $\bigcup_{n \in N}(\, R_\Omega(n) \cup R_\Psi(n)\,)$ represents the set of the labels corresponding to all the *risky actions*, i.e. the **`tell`** or **`retract`** actions that may lead to a policy violation. Conversely, if a policy does not appear in this set, there is no point in checking it, because it will always be obeyed at run time.

The procedure code follows that switches on the reference monitor on need (for clarity, we use a sugared version of ML$_{\text{CoDa}}$):

```
fun check_violation(e,l) =
   forall psi ∈ riskyΨ[l] do ask psi.();
   e();
   forall xi ∈ riskyΩ[l] ∪ riskyΨ[l] do ask xi.()
```

If there is a application policy associated with the label l we trigger the dispatching mechanism through the call `ask psi.()`. If the call fails then a policy violation is about to occur. In this case the computation is aborted or a recovery mechanism is invoked, if any. Otherwise the **`tell`** or **`retract`** is performed and every active policy, be it a context or an application one, is checked on the resulting context.

Our compilation schema needs to replace every **`tell`**`(e)`^{l} in the source code with the following:

```
check_violation(λ().tell(e)^l, l)
```

As expected, the same has to be done for every **`retract`**`(e)`^{l}. This is indeed what happens in the example above with the action corresponding to the **`retract`** $\texttt{F}_2^2$ that violates the policy ψ_1 (say that $\Lambda(2) = \text{l}_2$). After the instrumentation, that occurrence in the code will be `check_violation`$(\lambda()$.**`tell`** $(e)^{\text{l}_2}, l_2)$, and the execution will go stuck — unless there is a recovery mechanism, which we do not discuss here.

Note that, if the set of the risky actions is empty, then there is no need for checking, when calling either **`tell`** or **`retract`**.

7. Conclusions

Contributions We have addressed security issues in an adaptive framework, by extending and instrumenting ML$_{\text{CoDa}}$, a functional language introduced in [21,23] for adaptive programming. We also extended its two-step static analysis to deal with security. To the best of our knowledge, this is the first contribution to security *within* a linguistic approach to adaptivity, although its importance in this field is growing hot [56]. In summary, we have

- expressed and enforced *context-dependent* security policies using Datalog, originally employed by ML$_{\text{CoDa}}$ to deal with contexts. Policies are of two kinds: (i) context policies, a priori unknown to the applications, that protect the context from unwanted changes or accesses, and (ii) application policies, a priori unknown to the context manager, that protect the running code, and that can be nested and have a scope;
- extended the ML$_{\text{CoDa}}$ type and effect system of [22,23] for computing a type and a labelled abstract representation of the overall behaviour of an application, including the security policies it imposes and those it has to obey. Actually, an effect over-approximates the sequences of the possible dynamic actions over the context. The security-critical operations of the abstraction are labelled, so to link them with those in the code of the application, keeping track of the scope of the application policies;
- considerably enhanced the static analysis of [22,23] that guarantees an application to adapt to all the possible contexts arising at run time; now, our analysis also identifies the operations that may violate the context and the application policies in force. Recall that this step can only be done at load time, because the execution context is only known when the application is about to run, and thus our static analysis cannot be completed at compile time;
- defined a way to instrument at implementation time the code of an application, so as to incorporate in it an *adaptive reference monitor*, ready to stop executions when about to violate a policy to be enforced. When an application enters a new context, the results of the static analysis mentioned above are used to suitably drive the invocation of the monitor that is switched on and off if needed.

It is worth noting that ML$_{\text{CoDa}}$ itself required little extensions to be equipped with security policies and with mechanisms for checking and enforcing them. Indeed, policies are just Datalog clauses enforced by asking goals, and code is instrumented by using available constructs, namely behavioural variations and functions. Even though our proposal of a two-step static analysis is still a proof-of-concept, the underlying idea of complementing type-checking with the load time analysis, and the consequent instrumentation procedure appear to be well applicable to other adaptive programming paradigms.

Future work We are currently experimenting on our language, in particular on the usage of the static analysis on more realistic examples. We are using a prototypical implementation of ML$_{\text{CoDa}}$ that extends F# [15] and that is available together with some case studies.[2] To do that, we exploit the well-established metaprogramming mechanisms of F#, so to both minimise the learning cost for users and to avoid the need of any modification to the compiler and to the underlying .NET runtime. We also have a two-step type inference algorithm, and a naive construction of the evolution graph and of the reachability algorithm.

Besides a full implementation of ML$_{\text{CoDa}}$, and of an efficient tool for the static analysis and for the instrumentation, we plan to investigate recovery mechanisms appropriate for behavioural variations, to allow the user to undo some actions considered risky or sensible, and force the dispatching mechanism

[2] `https://github.com/vslab/fscoda`

to make different, alternative choices. Recovery mechanisms are obviously needed also to adapt applications that raise security failures. If the violated policy comes from the context, the easiest solution is abandoning that context, and possibly undoing the computation done so far. Nevertheless, this is not completely satisfying and makes it quite challenging to look for suitable and smarter recovery mechanisms. A long-term goal is extending these policies with quantitative information, e.g. statistical information about the usage of contexts, reliability of resources therein, so to rank contexts and to possibly suggest the user the ones guaranteeing more performance.

Related work Starting from the initial proposal of the COP paradigm by Costanza [20], several authors [31,1,37,5] mainly addressed the design and the implementation of concrete programming languages (see [6,62] for an overview). All these approaches describe the context as a stack of layers, which are elementary properties of the context. Layers can be activated/deactivated at run time, and those holding in the running context are determined by only considering the code of the application. Since behavioural variations are bound to layers, activating/deactivating a layer corresponds to activating/deactivating the corresponding behavioural variation. Some papers [19,30,34,3,4,35] just to cite a few, are also concerned with foundational aspects of various COP languages, often based on Java. They propose static (and dynamic) techniques for guaranteeing applications to perform well, e.g. that methods within behavioural variations are correctly invoked.

The usage of Datalog and of its rich predicates for representing the context is perhaps the main difference with the approaches mentioned above. Also, in $\mathrm{ML}_{\mathrm{CoDa}}$ behavioural variations are a first class, higher-order construct, that can then be referred to by identifiers, and used as parameters in functions. This fosters dynamic, compositional adaptation patterns, as well as reusable, modular code. Another difference with other COP languages is that the dispatching mechanism inspects the actual context that depends on both the application code and the“open” context unknown at development time.

As already remarked, we approached context-aware security from a formal linguistic viewpoint. To the best of our knowledge, in the literature there are few papers aiming at providing a uniform treatment of security and adaptivity. In the pure, foundational formalism of the Ambient Calculus [17] some papers, among which [53,24,13], studied properties concerning the way processes move within different environments, representing the topology of a network. However, in these proposals the context is a sub-set of ours, because it essentially consists of the locations where processes can be hosted.

The proposal in [56] is still along this line, but from a perspective more oriented towards software engineering. This paper can be seen as the starting point for the development of Ariadne [65], a tool for security in adaptive cyber-physical systems. Also in this case, the context is only focused on the topology of a cyber-physical scenario. Ariadne allows defining and maintaining a model of the running system, based on Bigraphical Reactive System [47]. When the system topology changes, an analysis at run time detects possible security risks, so to plan countermeasures.

A conceptual model of security contexts has been recently proposed in [36], that also takes into account social aspects. It is specifically designed for the specification, management and analysis of security issues in ubiquitous social systems.

Other approaches implement security mechanisms at different levels of the infrastructure, in the middleware [60] or in the interaction protocols [29]. Most of them address access control for resources, e.g. [68,33,69], and for smart things, e.g. [2,25]. In this strain, there are many papers on the ways of expressing context-aware access control. As an example, Role-Description Language (RDL) is a programming language for defining context-aware role-based access control policy [46]. The key idea is to activate or deactivate a role and the permissions granted by it depending on the context information. A RDL program receives as input context changes and outputs the members of each role. The proposal of [48] is similar to ours in that the rules are expressed in a logical fashion. The context information is represented through facts and logical rules, but the context itself is distributed among different principals, so that granting access to a resource requires a distributed deduction. Each principal is equipped with policies declaring which participants of the distributed system are considered trusted when resolving a particular query. An alternative approach to rule-based policies for context-aware access control is in [49], that proposes the notion of contextual graph to control the access to resources in a distributed environment. Along this line, Aspect-Oriented Programming and contexts are discussed in [50], as a pragmatic way to decoupling security concerns and business logic in Web Services.

The problem of context-aware security has often been faced in many specific domains of application. Below, we just mention a few papers, because they address issues that are orthogonal to our proposal. The work of [8,32] study authorisation in context-aware systems and propose mechanisms for granting authorisation transparently to a user operating in a smart environment. Access control for Android smartphones is addressed in [7], using a descriptive logic. There are some papers, among which the already cited [59,40], that study how protocols can be adapted to different operating environments in order to guarantee different levels of security.

References

[1] Achermann, F., Lumpe, M., Schneider, J., Nierstrasz, O.: PICCOLA-a small composition language. In: Formal methods for distributed processing. Cambridge University Press (2001)

[2] Al-Neyadi, F., Abawajy, J.: Context-based e-health system access control mechanism. Advances in information security and its application pp. 68–77 (2009)

[3] Aotani, T., Kamina, T., Masuhara, H.: Featherweight eventcj: a core calculus for a context-oriented language with event-based per-instance layer transition. In: Proc. of the 3rd International Workshop on Context-Oriented Programming (COP '11). pp. 1:1–1:7. ACM, New York, NY, USA (2011)

[4] Aotani, T., Kamina, T., Masuhara, H.: Unifying multiple layer activation mechanisms using one event sequence. In: Proceedings of 6th International Workshop on Context-Oriented Programming. pp. 2:1–2:6. COP'14, ACM, New York, NY, USA (2014)

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

[6] 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)

[7] Bai, G., Gu, L., Feng, T., Guo, Y., Chen, X.: Context-aware usage control for android. In: Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 50, pp. 326–343. Springer (2010)

[8] Bardram, J., Kjær, R.E., Pedersen, M.Ø.: Context-aware user authentication - supporting proximity-based login in pervasive computing. In: UbiComp 2003, Seattle, USA, Proceedings. Lecture Notes in Computer Science, vol. 2864, pp. 107–123. Springer (2003)

[9] Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security 17(5), 799–837 (2009), abridged version in Proc. of CSFW 2005, IEEE Press, 211-223

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

[11] Bodei, C., Degano, P., Galletta, L., Salvatori, F.: Linguistic mechanisms for context-aware security. In: Proc. of 11th International Colloquium on Theoretical Aspects of Computing. pp. 61–79. LNCS 8687, Springer (2014)

[12] 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)

[13] Bucur, D., Nielsen, M.: Secure data flow in a calculus for context awareness. In: Concurrency, Graphs and Models. Lecture Notes in Computer Science, vol. 5065, pp. 439–456. Springer (2008)

[14] 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: (ISSS'02). pp. 1–15. LNCS 2609, Springer (2003)

[15] Canciani, A., Degano, P., Ferrari, G.L., Galletta, L.: A context-oriented extension of F#. In: FOCLASA 2015. EPTCS, vol. 201, pp. 18–32 (2015)

[16] Cappaert, J.: Code Obfuscation Techniques for Software Protection. Ph.D. thesis, Katholieke Universität Loewen (2012), `https://www.cosic.esat.kuleuven.be/publications/thesis-199.pdf`

[17] Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. 240(1), 177–213 (2000)

[18] 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)

[19] Clarke, D., Sergey, I.: A semantics for context-oriented programming with layers. In: International Workshop on Context-Oriented Programming (COP '09). pp. 10:1–10:6. ACM, New York, NY, USA (2009)

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

[21] Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for COP. In: Proc. 6th International Workshop on Context-Oriented Programming. ACM Digital Library, doi: 10.1145/2637066.2637072 (2014)

[22] Degano, P., Ferrari, G.L., Galletta, L.: A two-phase static analysis for reliable adaptation. In: Proc. of 12th International Conference on Software Engineering and Formal Methods. pp. 347–362. LNCS 8702, Springer (2014)

[23] Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for adaptation: design, semantics and program analysis. IEEE Transactions on Software Engineering, 10.1109/TSE.2015.2496941. (2016)

[24] Degano, P., Levi, F., Bodei, C.: Safe ambients: Control flow analysis and security. In: 6th Asian Computing Science Conference, Malaysia, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1961, pp. 199–214. Springer (2000)

[25] Deng, M., Cock, D.D., Preneel, B.: Towards a cross-context identity management framework in e-health. Online Information Review 33(3), 422–442 (2009)

[26] 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 (2002)

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

[28] Galletta, L.: Adaptivity: linguistic mechanisms and static analysis techniques. Ph.D. thesis, University of Pisa (2014), `http://www.di.unipi.it/~galletta/phdThesis.pdf`

[29] 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)

[30] Hirschfeld, R., Igarashi, A., Masuhara, H.: ContextFJ: a minimal core calculus for context-oriented programming. In: Proc. of the 10th international workshop on Foundations of aspect-oriented languages. pp. 19–23. ACM (2011)

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

[32] Hulsebosch, R.J., Bargh, M.S., Lenzini, G., Ebben, P.W.G., Iacob, S.M.: Context sensitive adaptive authentication. In: EuroSSC 2007, Kendal, England, Proceedings. Lecture Notes in Computer Science, vol. 4793, pp. 93–109. Springer (2007)

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

[34] Igarashi, A., Hirschfeld, R., Masuhara, H.: A type system for dynamic layer composition. In: FOOL 2012. p. 13 (2012)

[35] Inoue, H., Igarashi, A., Appeltauer, M., Hirschfeld, R.: Towards type-safe JCop: A type system for layer inheritance and first-class layers. pp. 7:1–7:6. COP'14, ACM, New York, NY, USA (2014)

[36] Jovanovikj, V., Gabrijelcic, D., Klobucar, T.: A conceptual model of security context. Int. J. Inf. Sec. 13(6), 571–581 (2014)

[37] Kamina, T., Aotani, T., Masuhara, H.: EventCJ: a context-oriented programming language with declarative event-based context transition. In: Proc. of the 10 international conference on Aspect-oriented software development (AOSD '11). pp. 253–264. ACM (2011)

[38] Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)

[39] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In: Knudsen, J. (ed.) ECOOP 2001 — Object-Oriented Programming, Lecture Notes in Computer Science, vol. 2072, pp. 327–354. Springer Berlin Heidelberg (2001)

[40] Ksiezopolski, B., Kotulski, Z.: Adaptable security mechanism for dynamic environments. Computers & Security 26(3), 246–255 (2007)

[41] 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 (PADL '03). pp. 58–73. LNCS 2562, Springer (2003)

[42] Ligatti, J., Walker, D., Zdancewic, S.: A type-theoretic interpretation of pointcuts and advice. Science of Computer Programming 63(3), 240 – 266 (2006)

[43] 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)

[44] MacDonal, N.: The future of information security is context awere and adaptive. Tech. rep., Gartner RAS (2010)

[45] Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6), 3–14 (Oct 1996)

[46] Masone, C., Kotz, A.D.: Role definition language (rdl): A language to describe context-aware roles. Tech. rep. (2002)

[47] Milner, R.: Bigraphical reactive systems. In: CONCUR 2001, Aalborg, Denmark, Proceedings. Lecture Notes in Computer Science, vol. 2154, pp. 16–35. Springer (2001)

[48] Minami, K., Kotz, D.: Secure context-sensitive authorization. Pervasive and Mobile Computing 1(1), 123–156 (2005)

[49] Mostéfaoui, G.K., Brézillon, P.: Context-based constraints in security: Motivations and first approach. Electr. Notes Theor. Comput. Sci. 146(1), 85–100 (2006)

[50] Mostéfaoui, G.K., Maamar, Z., Narendra, N.C.: SC-WS: A context-based, aspect-oriented approach for handling security concerns in web services. IJOCI 4(2), 31–44 (2014)

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

[52] Nanevski, A., Banerjee, A., Garg, D.: Dependent type theory for verification of information flow and access control policies. ACM Trans. Program. Lang. Syst. 35(2), 6 (2013)

[53] Nielson, F., Riis Nielson, H., Hansen, R.R., Jensen, J.G.: Validating firewalls in mobile ambients. In: CONCUR '99, Eindhoven, The Netherlands, Proceedings. Lecture Notes in Computer Science, vol. 1664, pp. 463–477. Springer (1999)

[54] 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. pp. 223–244. LNCS 2566, Springer (2002)

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

[56] Pasquale, L., Ghezzi, C., Menghi, C., Tsigkanos, C., Nuseibeh, B.: Topology aware adaptive security. In: Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 43–48. SEAMS 2014, ACM, New York, NY, USA (2014)

[57] Perera, C., Zaslavsky, A.B., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. IEEE Communications Surveys and Tutorials 16(1), 414–454 (2014)

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

[59] Pierson, L.G., Witzke, E.L., Bean, M.O., Trombley, G.J.: Context-agile encryption for high speed communication networks. Computer Communication Review 29(1), 35–49 (1999)

[60] 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)

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

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

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

[64] Spinczyk, O., Gal, A., Schröder-Preikschat, W.: Aspectc++: An aspect-oriented extension to the c++ programming language. pp. 53–60. CRPIT '02, Australian Computer Society, Inc., Darlinghurst, Australia, Australia (2002)

[65] Tsigkanos, C., Pasquale, L., Ghezzi, C., Nuseibeh, B.: Ariadne: Topology aware adaptive security for cyber-physical systems. In: 37th IEEE/ACM International Conference on Software Engineering, Florence, Italy, Volume 2. pp. 729–732. IEEE (2015)

[66] Walker, D., Zdancewic, S., Ligatti, J.: A Theory of Aspects. SIGPLAN Not. 38(9), 127–139 (Aug 2003)

[67] Wand, M., Kiczales, G., Dutchyn, C.: A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming. ACM Trans. Program. Lang. Syst. 26(5), 890–910 (Sep 2004)

[68] 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)

[69] 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

Below, we present the proofs of the theorems establishing the correctness of our proposal, where we feel free to omit the labels whenever immaterial.

A. Properties of the Type and Effect System

Below we prove Theorem 4.1, Theorem 4.2 and Corollary 1. We start giving some lemmas and definitions useful for the formal development.

Definition A.1 (Capture avoiding substitutions)**.** Given the expressions e, e' and the variable x we define $e\{e'/x\}$ as follows

$$c\{e'/x\} = c$$

$$F\{e'/x\} = F$$

$$(\lambda_f x'.e)\{e'/x\} = \lambda_f x'.e\{e'/x\}$$

$$\text{if } f \neq x \wedge x' \neq x \wedge f, x' \notin FV(e')$$

$$(x')\{G_1.e_1, \ldots, G_n.e_n\}\{e'/x\} = (x')\{G_1.e_1\{e'/x\}, \ldots, G_n.e_n\{e'/x\}\}$$

$$\text{if } x \neq x' \wedge x \in \bigcup_{i \in \{1,\ldots,n\}} FV(G_i) \wedge$$

$$\left(\{x'\} \cup \bigcup_{i \in \{1,\ldots,n\}} FV(G_i)\right) \cap FV(e') = \emptyset$$

$$x\{e'/x\} = e'$$

$$x'\{e'/x\} = x' \qquad \text{if } x \neq x'$$

$$(e_1\, e_2)\{e'/x\} = e_1\{e'/x\}\, e_2\{e'/x\}$$

$$(e_1\, op\, e_2)\{e'/x\} = e_1\{e'/x\}\, op\, e_2\{e'/x\}$$

$$(\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3)\{e'/x\} = \texttt{if}\, e_1\{e'/x\}\, \texttt{then}\, e_2\{e'/x\}\, \texttt{else}\, e_3\{e'/x\}$$

$$(\texttt{tell}^l(e))\{e'/x\} = \texttt{tell}^l(e\{e'/x\})$$

$$(\texttt{retract}^l(e))\{e'/x\} = \texttt{retract}^l(e\{e'/x\})$$

$$(e_1 \cup e_2)\{e'/x\} = e_1\{e'/x\} \cup e_2\{e'/x\}$$

$$\#(e_1,\, e_2)\{e'/x\} = \#(e_1\{e'/x\},\, e_2\{e'/x\})$$

$$(\texttt{let}\, x' \;=\; e_1\, \texttt{in}\, e_2)\{e'/x\} = \texttt{let}\, x' \;=\; e_1\{e'/x\}\, \texttt{in}\, e_2\{e'/x\} \qquad \text{if } x \neq x' \wedge x' \in FV(e')$$

$$(\texttt{dlet}\, \tilde{x} \;=\; e_1\, \texttt{when}\, G\, \texttt{in}\, e_2)\{e'/x\} = \texttt{dlet}\, \tilde{x} \;=\; e_1\{e'/x\}\, \texttt{when}\, G\, \texttt{in}\, e_2\{e'/x\}$$

$$\text{if } x \notin FV(G) \wedge FV(G) \cap FV(e') = \emptyset$$

$$(\psi^l[e])\{e'/x\} = \psi^l[e\{e'/x\}].$$

Lemma A.1. *If* $\Gamma \vdash \rho : K$ *and* $K \sqsubseteq K'$ *then* $\Gamma \vdash \rho : K'$.

Proof. The thesis follows from Definition 4.2 and that of $K \sqsubseteq K'$. □

In the following we denote with $K_{\tilde{x}} = K \backslash (\tilde{x}, \tau, \Delta, \Lambda)$

Lemma A.2. *Given* K *and a parameter* $\tilde{x}$

1. *if* $\tilde{x} \notin K$ *then* $K \sqsubseteq K_{\tilde{x}}, (\tilde{x}, \tau, \Delta, \Lambda)$ *for all* τ, Δ, *and* Λ
2. *if* $K(\tilde{x}) = (\tau, \Delta, \Lambda)$ *then* $K \sqsubseteq K_{\tilde{x}}, (\tilde{x}, \tau_1, \Delta_1 \otimes \Delta, \Lambda_1)$ *for all* $\tau \leq \tau_1$, Δ_1 *and* $\Lambda \sqsubseteq \Lambda_1$

Proof. The thesis follows by using the definition of $K \sqsubseteq K'$. □

Lemma A.3. *If* $\Gamma \vdash \rho : K$ *and* G *and* e *are such that* $\gamma(G) = \overrightarrow{y} : \overrightarrow{\tau}$ *and* $\Gamma, \overrightarrow{y} : \overrightarrow{\tau}; K \vdash e : \tau \triangleright H; \Lambda$

1. *for all* $\tilde{x} \notin dom(\rho)$ *then* $\Gamma \vdash \rho[G.e/\tilde{x}] : K_{\tilde{x}}, (\tilde{x}, \tau, askG.H, \Lambda)$
2. *if* $\rho(\tilde{x}) = G'_1.e'_1, \ldots G'_n.e'_n$ *and* $K(\tilde{x}) = (\tau, \Delta, \Lambda')$ *then* $\Gamma \vdash \rho[G.e, \rho(\tilde{x})/\tilde{x}] : K_{\tilde{x}}, (\tilde{x}, \tau, askG.H \otimes \Delta, \Lambda \uplus \Lambda')$.

Proof. The thesis follows by using the definition 4.2 and that of $K \sqsubseteq K'$. □

Lemma A.4. *If* $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ *and* Γ' *and* K' *are permutation of* Γ *and* K *respectively, then* $\Gamma'; K' \vdash e : \tau \triangleright H; \Lambda$.

Proof. Straightforward induction on typing derivations. □

Lemma A.5 (Weakening)**.**

1. *if* $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ *and* x *is a variable* $x \notin dom(\Gamma)$ *then* $\Gamma, x : \tau'; K \vdash e : \tau \triangleright H; \Lambda$ *for some* τ'.
2. *if* $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ *and* $\tilde{x}$ *is a parameter* $\tilde{x} \notin dom(K)$ *then* $\Gamma; K, (\tilde{x}, \tau', \Delta', \Lambda') \vdash e : \tau \triangleright H; \Lambda$ *for some* τ', Δ' *and* Λ'.

Proof. By a standard induction on the depth of the derivations. □

Lemma A.6 (Inclusion)**.**

1. *If* $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma'; K \vdash e : \tau \triangleright H; \Lambda$.
2. *If* $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ *and* $K \sqsubseteq K'$ *then* $\Gamma; K' \vdash e : \tau \triangleright H; \Lambda$.

Proof.

1. Since $\Gamma \subseteq \Gamma'$ there exists a set of binding $\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \subseteq \Gamma'$ such that $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n = \Gamma'$, so by applying n times Lemma A.5 the thesis holds.
2. Similar to previous case.

□

Lemma A.7 (Canonical form)**.** *If* v *is a value such that*

1. $\Gamma; K \vdash v \,:\, \tau_c \rhd H; \Lambda$ *then* $v = c$
2. $\Gamma; K \vdash v \,:\, \tau_1 \xrightarrow{K'|H'} \tau_2 \rhd H; \Lambda$ *then* $v = \lambda_f x.e$
3. $\Gamma; K \vdash v \,:\, \tau_1 \overset{K'|\Delta}{\Longrightarrow} \tau_2 \rhd H; \Lambda$ *then* $v = (x)\{Va\}$
4. $\Gamma; K \vdash v \,:\, \mathit{fact}_{\{F_1,\dots,F_m\}} \rhd H; \Lambda$ *then* $v \in \{F_1, \dots, F_m\}$

Proof.

1. Values can only have four forms: c, $(x)\{Va\}$, $\lambda_f x.e$ and F. If v has type τ_c the only rule which we can apply is (TCONST) hence $v = c$.
2. It follow from a reasoning similar to (1).
3. It follow from a reasoning similar to (1).
4. The fact type with annotations $\{F_1, \dots, F_n\}$ can be only deduced by applying the (TSUB) rule, starting from a type annotated with a singleton set $\{F\}$ for some $F \in \{F_1, \dots, F_n\}$. So this type can be obtained by (TFACT) rule only, hence $v = F$.

□

Lemma A.8 (Decomposition Lemma)**.**

1. *If* $\Gamma; K \vdash \lambda_f x.e \,:\, \tau_1 \xrightarrow{K'|H;\Lambda} \tau_2 \rhd H'; \Lambda'$ *and* $K' \sqsubseteq K$ *then* $\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K \vdash e \,:\, \tau_2 \rhd H; \Lambda$
2. *If* $\Gamma; K \vdash (x)\{G_1.e_1, \dots, G_n.e_n\} \,:\, \tau_1 \overset{K'|\Delta;\Lambda}{\Longrightarrow} \tau_2 \rhd H'; \Lambda'$ *and* $K' \sqsubseteq K$ *and* $\Delta = \bigotimes_{i \in \{1,\dots,n\}} \mathit{ask}\, G_i.H_i$ *then* $\forall i \in \{1, \dots, n\}$ $\Gamma, x : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K \vdash e_i \,:\, \tau_2 \rhd H_i; \Lambda$ *where* $\overrightarrow{y_i} : \overrightarrow{\tau_i} = \gamma(G_i)$

Proof.

1. By the premise of the rule (TABS) we know that $\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H;\Lambda} \tau_2; K' \vdash e \,:\, \tau_2 \rhd H; \Lambda$. Since, $K' \sqsubseteq K$, the thesis follows by Lemma A.6.
2. By the premise of the rule (TVARIATION) we know that $\forall i \in \{1, \dots, n\}$ $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K' \vdash e_i \,:\, \tau_2 \rhd H_i; \Lambda$ and $\overrightarrow{y_i} : \overrightarrow{\tau_i} = \gamma(G_i)$ and $\Delta = \bigotimes_{i \in \{1,\dots,n\}} \mathit{ask}\, G_i.H_i$. Since $K' \sqsubseteq K$ the thesis follows by Lemma A.6(2). □

Lemma A.9 (Substitution)**.** *If* $\Gamma, x : \tau'; K \vdash e \,:\, \tau \rhd H; \Lambda$ *and* $\Gamma; K \vdash v \,:\, \tau' \rhd \epsilon; \bot$ *then* $\Gamma; K \vdash e\{v/x\} \,:\, \tau \rhd H; \Lambda$.

Proof. By induction on the depth of the typing derivation, and then by cases on the last rule applied.

– rule (TTELL)
 By the premise of the rule, we know that $\Gamma, x : \tau'; K \vdash e \,:\, \mathit{fact}_\phi \rhd H; \Lambda$ holds. By using the induction hypothesis, we can claim that $\Gamma; K \vdash \mathtt{tell}(e\{v/x\}) \,:\, \tau \rhd H; \Lambda$ and by Definition A.1 we can conclude that $\Gamma; K \vdash (\mathtt{tell}(e))\{v/x\} \,:\, \tau \rhd H; \Lambda$.

– rule (TRETRACT)
 Similar to the case (TTELL)

- rule (TAPPEND)
  By the premise of the rule we know that $\Gamma, x : \tau'; K \vdash e_i : \tau_1 \xRightarrow{K'|\Delta_i} \tau_2 \triangleright H_i; \Lambda_i$ for $i \in \{1, 2\}$ holds. By the inductive hypothesis we can claim that $\Gamma; K \vdash e_1\{v/x\} \cup e_2\{v/x\} : \tau \triangleright H; \Lambda$ holds. By Definition A.1 we conclude $\Gamma; K \vdash (e_1 \cup e_2)\{v/x\} : \tau \triangleright H; \Lambda$.
- rule (TVAPP)
  By the premise of the rule we know that $\Gamma, x : \tau'; K \vdash e_1 : \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright H_1; \Lambda_1$ and $\Gamma, x : \tau'; K \vdash e_2 : \tau_1 \triangleright H_2; \Lambda_2$ and $K' \sqsubseteq K$. By using the induction hypothesis we can claim that $\Gamma; K \vdash \#(e_1\{v/x\}, e_2\{v/x\}) : \tau \triangleright H; \Lambda$ holds and by Definition 4.2 we can conclude that $\Gamma; K \vdash \#(e_1, e_2)\{v/x\} : \tau \triangleright H; \Lambda$.
- rule (TVARIATION)
  By the premise of the rule (TVARIATION) we know that $\forall i \in \{1, \ldots, n\}$ $\Gamma, x : \tau', x' : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K' \vdash e_i : \tau_2 \triangleright H_i; \Lambda_i$ where $\overrightarrow{y_i} : \overrightarrow{\tau_i} = \gamma(G_i)$, $\Delta = \bigotimes_{i \in \{1,\ldots,n\}} ask\, G_i.H_i$. By Lemma A.4 $\forall i \in \{1, \ldots, n\}$ $\Gamma, x' : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}, x : \tau'; K' \vdash e_i : \tau_2 \triangleright H_i; \Lambda_i$. By using the induction hypothesis and the rule (TVARIATION) we can claim that $\Gamma; K \vdash (x')\{G_1.e_1\{v/x\}, \ldots, G_n.e_n\{v/x\}\} : \tau \triangleright H; \Lambda$ and by Definition A.1 we conclude $\Gamma; K \vdash (x')\{G_1.e_1, \ldots, G_n.e_n\}\{v/x\} : \tau \triangleright H; \Lambda$.
- rule (TDLET)
  By the precondition of the rule (TDLET) we know that $\Gamma, x : \tau', \overrightarrow{y} : \overrightarrow{\tau}; K \vdash e_1 : \tau_1 \triangleright H_1; \Lambda_1$ and $\Gamma, x : \tau'; K, (\tilde{x}, \tau_1, \Delta, \Lambda_3) \vdash e_2 : \tau \triangleright H_2; \Lambda_2$ with $\overrightarrow{y} : \overrightarrow{\tau} = \gamma(G)$. By Lemma A.4 $\Gamma, \overrightarrow{y} : \overrightarrow{\tau}, x : \tau'; K \vdash e_1 : \tau_1 \triangleright H_1; \Lambda_1$. By using the induction hypothesis we can claim that $\Gamma; K \vdash$ $\mathtt{dlet}\, \tilde{x} = e_1\{v/x\}\, \mathtt{when}\, G\, \mathtt{in}\, e_2\{v/x\} : \tau \triangleright H_2; \Lambda_2$ and by Definition A.1 $\Gamma; K \vdash (\mathtt{dlet}\, \tilde{x} = e_1\, \mathtt{when}\, G\, \mathtt{in}\, e_2)\{v/x\} : \tau \triangleright H_2; \Lambda_2$.
- rule (TCONST), (TFACT), (TDVAR) Since $e\{v/x\} = e$ by Definition A.1 the result $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ is immediate, when $e = c$, $e = F$ and $e = \tilde{x}$.
- The other cases are standard.

□

Lemma A.10. *If* $\Gamma, x : \tau'; K \vdash e : \tau \triangleright H; \Lambda$ *and* z *is a variable such that* $z \notin FV(e)$ *and* z *does not occur in* Γ *then* $\Gamma, z : \tau'; K \vdash e\{z/x\} : \tau \triangleright H; \Lambda$.

Proof. Similar to that of Lemma A.9. □

Lemma A.11. *If* $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ *then* $\Gamma; K \vdash v : \tau \triangleright \epsilon; \bot$

Proof. Any derivation for the judgement $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ starts from the axiom $\Gamma; K \vdash v : \tau' \triangleright \epsilon; \bot$ for some τ', and only contains suitable applications of rule (TSUB) for enlarging the type, the effect and the labelling environment.

□

Lemma A.12. *If* $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ *then there exist* $H_1, \ldots, H_n$ *such that* $H = \epsilon + \Sigma_{i=1}^{n} H_i$

Proof. Any derivation for the judgement $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ starts from the axiom $\Gamma; K \vdash v : \tau' \triangleright \epsilon; \bot$ for some τ'. Then suitable applications of rule (TSUB) for enlarging the type, the effect and the labelling environment, since $H \sqsubseteq \epsilon + \Sigma_{i=1}^{n} H_i$ by definition. □

Lemma A.13. *If* $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ *then for all* K' *we have that* $\Gamma; K' \vdash v : \tau \triangleright H; \Lambda$.

Proof. By induction on the depth of the typing derivation. □

Lemma A.14. *If* $C, H \dashrightarrow^* C', H'$ *then*

1. $C, H \cdot H'' \dashrightarrow^* C', H' \cdot H''$ *for all* H'',
2. *for all* C *such that* $C \not\models G_j$ *for* $j \in \{1, \ldots, i-1\}$ *and* $C \models G_i$ *and* $H_i = H$, *it is* $C, \bigotimes_{k \in \{1,\ldots,n\}} askG_k.H_k \dashrightarrow^* C', H'$.

Proof. Item (1) is immediate by applying the rule for summation. Item (ii) follows by induction on the length of the computation $C, H \dashrightarrow^* C', H'$, applying the rule for $\cdot$. For proving item (iii), we just apply i times the rule $C \not\models G_j$ for abstract variation. □

The proof of the following three properties follows immediately by definition $\sqsubseteq$ and by the semantics of history expressions.

Property A.15. *Let* H *be a history expression then* $\epsilon \cdot H = H$.

Property A.16. *Let* H_1, H_2, H_3 *be history expressions, then it holds* $(H_1 + H_2) \cdot H_3 = H_1 \cdot H_3 + H_2 \cdot H_3$.

Property A.17. *If* $H \sqsubseteq H'$ *then* $H \cdot H'' \sqsubseteq H' \cdot H''$ *and* $\psi^l[H] \sqsubseteq \psi^l[H']$.

Theorem 4.1 (Preservation).
Let e *be a closed expression; and let* ρ *be a dynamic environment such that* $dom(\rho)$ *includes the set of parameters of* e *and* $\Gamma \vdash \rho : K$.
If $\Gamma;\ K \vdash e_s : \tau \triangleright H_s; \Lambda_s$ *and* $\rho \vdash C,\ e_s \to C',\ e'_s$, *then* $\Gamma;\ K \vdash e'_s : \tau \triangleright H'_s; \Lambda'_s$, $C, H_s \dashrightarrow^* C',\ H''$ *for some* $H'' \sqsubseteq H'_s$ *and* $\Lambda'_s \sqsubseteq \Lambda_s$.

Proof. By induction on the depth of the typing derivation and then by cases on the last rule applied.
In the proof we implicitly use the fact that $H \sqsubseteq H$ for each H (except for the case TSUB).

- rule (TVARIATION) or (TCONST) or (TFACT) or (TABS) or (TVAR)
  In this case we know that e_s is a value (or a variable in the case (TVAR)), then for no e'_s it holds $\rho \vdash C,\ e_s \to C',\ e'_s$, so the theorem holds vacuously.
- rule (TTELL)
  We know that $e_s = \texttt{tell}(e')^l$ for some e' and also by the (TTELL) premise that $\Gamma; K \vdash e' : fact_\phi \triangleright H; \Lambda$ holds and $H_s = (H \cdot \sum_{F \in \phi} tell\, F^{l_i})^l$. We have only two rules by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.

  * rule (TELL1)
    We know that e' is an expression and $e'_s = \texttt{tell}(e'')^l$ and $\rho \vdash C,\ e' \to C',\ e''$ and there is in our derivation a subderivation with conclusion $\Gamma; K \vdash e' : fact_\phi \triangleright H; \Lambda$. By the induction hypothesis $\Gamma; K \vdash e'' : fact_\phi \triangleright H''; \Lambda''$, $C,\ H \dashrightarrow^\star C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H''$ and $\Lambda'' \sqsubseteq \Lambda$. By using the rule (TTELL) we can conclude that $\Gamma; K \vdash e'_s : unit \triangleright H'_s; \Lambda'_s$, $H'_s = (H'' \cdot \sum_{F \in \phi} tell\, F^{l_i})^l$ and $\Lambda'_s = \Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l]$. Lemma A.14 now suffices for establishing that $C, H \cdot \sum_{F \in \phi} tell\, F^{l_i} \dashrightarrow^\star C', \overline{H} \cdot \sum_{F \in \phi} tell\, F^{l_i}$ and by Property A.17 $\overline{H} \cdot \sum_{F \in \phi} tell\, F^{l_i} \sqsubseteq H'' \cdot \sum_{F \in \phi} tell\, F^{l_i}$. Since $\Lambda'_s = \Lambda'' \uplus_{F_i \in \phi} [l_i \mapsto l] \sqsubseteq \Lambda \uplus_{F_i \in \phi} [l_i \mapsto l] = \Lambda_s$ the thesis follows.

  - rule (TELL2)
    We now that $e' = F$, $e'_s = ()$ and $C' = C \cup \{F\}$. We have to prove that $\Gamma; K \vdash e'_s \ : \ unit \triangleright H'_s; \Lambda'_s$, but from the rule (TCONST) we know that this holds with $H'_s = \epsilon$ and $\Lambda'_s = \bot \sqsubseteq \Lambda_s$. It remains to show that $C, H_s \ \rightarrow^* \ C', H'_s$. From Lemma A.12 we know $H_s = \epsilon + \Sigma_{i=1}^{n} H_i$. Then, $C, (\epsilon + \Sigma_{i=1}^{n} H_i) \cdot \Sigma_{F \in \phi}\, tell\, F^{l_i} \rightarrow C, \epsilon \cdot \Sigma_{F \in \phi}\, tell\, F^{l_i} \rightarrow \Sigma_{F \in \phi}\, tell\, F^{l_i} \rightarrow C, tell\, F^{l} \rightarrow C \cup \{F\}, \epsilon = C', H'_s$.

- rule (TRETRACT)
  Similar to (TTELL) rule ($retract$ substitutes $tell$)
- rule (TAPPEND)
  We know $e_s = e_1 \cup e_2$ and $\tau = \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2; \Lambda_3 \otimes \Lambda_4} \tau_2$ and $H_s = H_1 \cdot H_2$ and $\Lambda_s = \Lambda_1 \uplus \Lambda_2$, and also by the premise of (TAPPEND) that $\Gamma; K \vdash e_1 \ : \ \tau_1 \xRightarrow{K'|\Delta_1; \Lambda_3} \tau_2 \triangleright H_1; \Lambda_1$ and $\Gamma; K \vdash e_2 \ : \ \tau_1 \xRightarrow{K'|\Delta_2; \Lambda_4} \tau_2 \triangleright H_2; \Lambda_2$ hold. There are three rules only by which $\rho \vdash C, \, e_s \rightarrow C', \, e'_s$ can be derived.

  - rule (APPEND1)
    We know that e_1 and e_2 are not values and $e'_s = e'_1 \cup e_2$. By applying the induction hypothesis $\Gamma; K \vdash e'_1 \ : \ \tau_1 \xRightarrow{K'|\Delta_1} \tau_2 \triangleright H'_1; \Lambda'_1$ with $C, H_1 \ \rightarrow^* \ C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$ and $\Lambda'_1 \sqsubseteq \Lambda_1$. By applying the (TAPPEND) rule we can conclude that $\Gamma; K \vdash e'_1 \cup e_2 \ : \ \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright H'_1 \cdot H_2; \Lambda'_1 \uplus \Lambda_2$. The thesis follows by applying Lemma A.14 and Property A.17.
  - rule (APPEND2)
    We know that $e'_s = (x)\{Va_1\} \cup e'_2$. By applying the induction hypothesis $\Gamma; K \vdash e'_2 \ : \ \tau_1 \xRightarrow{K'|\Delta_2; \Lambda_4} \tau_2 \triangleright H'_2; \Lambda'_2$ with $C, \, H_2 \ \rightarrow^* \ C', \, \overline{H}$ for some $\overline{H} \sqsubseteq H'_2$ and $\Lambda'_2 \sqsubseteq \Lambda_2$. By the rule (TVARIATION) we know that $\Gamma; K \vdash (x)\{Va_1\} \ : \ \tau_1 \xRightarrow{K'|\Delta_1} \tau_2 \triangleright \epsilon; \bot$ and by applying the rule (TAPPEND) we can claim that $\Gamma; K \vdash (x)\{Va_1\} \cup e'_2 \ : \ \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright \epsilon \cdot H'_2; \bot \uplus \Lambda'_2$. Then $\bot \uplus \Lambda'_2 \sqsubseteq \Lambda_1 \uplus \Lambda_2$ and $\epsilon \cdot H'_2 = H'_2 = H'_s$ by Property A.15. By Lemma A.12 we know $H_1 = (\epsilon + \Sigma_{i=1}^{n} H_i)$, then $C, H_s = C, H_1 \cdot H_2 \rightarrow C, \epsilon \cdot H_2 \rightarrow C, H_2 \rightarrow^* \ C', \overline{H}$, proving the thesis since $\overline{H} \sqsubseteq H'_s$.
  - rule (APPEND3)
    We know that e_s is

$$(x)\{G_1.e_1, \ldots, \, G_n.e_n\} \cup (y)\{G'_1.e'_1, \ldots, \, G'_m.e'_m\}$$

and that e'_s is

$$\begin{aligned}&(z)\{G_1.e_1\{z/x\}, \ldots, \, G_n.e_n\{z/y\},\\ &\qquad G'_1.e'_1\{z/y\}, \ldots, \, G'_m.e'_m\{z/x\}\}.\end{aligned}$$

By the premise of the rule (TVARIATION), we also know that $\forall i \in \{1, \ldots, n\}$ we have $\Gamma, \, x \ : \ \tau_1, \overrightarrow{y_i} \ : \ \overrightarrow{\tau_i}; K' \vdash e_i \ : \ \tau_2 \triangleright H_i; \Lambda_i$ (recall that $\Lambda_3 = \uplus_i \Lambda_i$) and $\forall j \in \{1, \ldots, m\}$ we have $\Gamma, y \ : \ \tau_1, \overrightarrow{y_j} : \overrightarrow{\tau_j}; K' \vdash e'_j \ : \ \tau_2 \triangleright H_j; \Lambda_j$ (recall that $\Lambda_4 = \uplus_j \Lambda_j$). By Lemma A.10 it holds that $\forall i \in \{1, \ldots, n\} \ \Gamma, \, z \ : \ \tau_1, \overrightarrow{y_i} \ : \ \overrightarrow{\tau_i}; K' \vdash e_i\{z/x\} \ : \ \tau_2 \triangleright H_i; \Lambda_i$ and $\forall j \in \{1, \ldots, m\} \ \Gamma, z \ : \ \tau_1, \overrightarrow{y_j} : \overrightarrow{\tau_j}; K' \vdash e'_j\{z/x\} \ : \ \tau_2 \triangleright H_j; \Lambda_j$. So by applying the rule (TVARIATION) for all judgements indexed by i and j we can conclude that $\Gamma; K \vdash e'_s \ : \ \tau_1 \xRightarrow{K'|\Delta_1 \otimes \Delta_2; \Lambda_3 \uplus \Lambda_4} \tau_2 \triangleright \epsilon; \bot = H'_s; \Lambda'_s$. By applying twice Lemma A.12 we have $H_j = (\epsilon + \Sigma_{i=1}^n H_i)$ for $j \in \{1, 2\}$. Then, the thesis follows because $C, H_s = C, H_1 \cdot H_2 \to^* C, H_2 \to C, \epsilon$.

- rule (TVAPP)
  We know that $\Gamma; K \vdash e_1 \ : \ \tau_1 \xRightarrow{K'|\Delta;\Lambda_3} \tau_2 \triangleright H_1; \Lambda_1, \Gamma; K \vdash e_2 \ : \ \tau_1 \triangleright H_2; \Lambda_2$ and $K' \sqsubseteq K$ hold by (TVAPP) premises. There are three rules only by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.

  * rule (VAPP1)
    We know that $e'_s = \#(e'_1,\ e_2)$. By the induction hypothesis $\Gamma; K \vdash e'_1 \ : \ \tau_1 \xRightarrow{K'|\Delta;\Lambda_3} \tau_2 \triangleright H'_1; \Lambda'_1$ with $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$. By (TVAPP) rule we have $\Gamma; K \vdash e'_s \ : \ \tau_2 \triangleright H'_1 \cdot H_2 \cdot \Delta; \Lambda'_1 \uplus \Lambda_2 \uplus \Lambda_3$. By Lemma A.14 we can conclude $C, H_1 \cdot H_2 \cdot \Delta \to^* C', \overline{H} \cdot H_2 \cdot \Delta$ and the thesis follows by Property A.17 and because $\Lambda'_1 \uplus \Lambda_2 \uplus \Lambda_3 \sqsubseteq \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3$.
  * rule (VAPP2)
    We know that $e'_s = \#((x)\{Va\},\ e'_2)$. By using Lemma A.11 we have $\Gamma; K \vdash (x)\{Va\} \ : \ \tau_1 \xRightarrow{K'|\Delta;\Lambda_3} \tau_2 \triangleright \epsilon; \bot$ and by the induction hypothesis $\Gamma; K \vdash e'_2 \ : \ \tau_1 \triangleright H'_2; \Lambda'_2$ with $C,\ H_2 \to^* C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H'_2$ and $\Lambda'_2 \sqsubseteq \Lambda_2$. By (TVAPP) and Property A.15 $\Gamma; K \vdash e'_s \ : \ \tau_2 \triangleright \epsilon \cdot H'_2 \cdot \Delta; \bot \uplus \Lambda'_2 \uplus \Lambda_3 = H'_2 \cdot \Delta; \Lambda'_s$ holds. By Lemma A.12 we have $H_1 = (\epsilon + \Sigma_{i=1}^n H_i)$, then $C, H_1 \cdot H_2 \cdot \Delta \to C, \epsilon \cdot H_2 \cdot \Delta \to C, H_2 \cdot \Delta$. By Lemma A.14 we have $C, H_2 \cdot \Delta \to^* C', \overline{H} \cdot \Delta$ and Property A.17 and $\Lambda'_s = \bot \uplus \Lambda'_2 \uplus \Lambda_3 \sqsubseteq \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3 = \Lambda_s$ prove the thesis.
  * rule (VAPP3)
    We know that $e_s = \#((x)\{Va\},\ v)$ where $Va = G_1.e_1, \ldots, G_n.e_n$, $e'_s = e_j\{v/x,\ \overrightarrow{c}/\overrightarrow{y}\}$ for $j \in \{1, \ldots, n\}$ and $\rho \vdash C,\ e_s \to C,\ e'_s$. From our hypothesis and from Lemma A.8(2) we have that for all $i \in \{1, \ldots, n\}$ it holds $\Gamma, x \ : \ \tau_1, \overrightarrow{y_i} \ : \ \overrightarrow{t_i}; K \vdash e_i \ : \ \tau_2 \triangleright H_i; \Lambda_3$. By Lemma A.11 we also know that $\Gamma; K \vdash v \ : \ \tau_1 \triangleright \epsilon; \bot$. So by Lemma A.9 we have that for $i \in \{1, \ldots, n\}$ $\Gamma; K \vdash e_i\{v/x, \overrightarrow{\tau}/\overrightarrow{y}\} \ : \ \tau \triangleright H_i; \Lambda_3$. By Lemma A.12 we have $H_j = \epsilon$ for $j \in \{1, 2\}$, then $C, H_1 \cdot H_2 \cdot \Delta \to C, \epsilon \cdot H_2 \cdot \Delta \to C, H_2 \cdot \Delta \to C, \epsilon \cdot \Delta \to C, \Delta$. The thesis follows by using Lemma A.14.

- rule (TDLET)
  If the last rule in the derivation is (TDLET) we know that there is a subderivation with conclusions $\gamma(G) = \overrightarrow{y} : \overrightarrow{\tau}$ and $\Gamma, \overrightarrow{y} : \overrightarrow{\tau}; K \vdash e_1 \ : \ \tau_1 \triangleright H_1; \Lambda_1$ and $\Gamma; K_{\tilde{x}}, (\tilde{x},\ \tau_1,\ \Delta',\ \Lambda') \vdash e_2 \ : \ \tau \triangleright H_2; \Lambda_2$ with $\Delta' = askG.H_1 \otimes fail$ $\Lambda' = \Lambda_1$ when $\tilde{x} \notin dom(K)$ or $\Delta' = askG.H_1 \otimes \Delta$ and $\Lambda' = \Lambda_1 \uplus \Lambda$ when $K(\tilde{x}) = (\tau_1,\ \Delta,\ \Lambda)$. There are two rules by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.

  * rule (DLET1)
    We know that $e'_s = \texttt{dlet}\, \tilde{x} = e_1\, \texttt{when}\, G\, \texttt{in}\, e'_2$ and $\rho' \vdash C, e_2 \to C', e'_2$ with $\rho' = \rho[G.e_1, \rho(\tilde{x})/\tilde{x}]$. By Lemma A.1 $\Gamma \vdash \rho \ : \ K'$ with $K' = K_{\tilde{x}}, (\tilde{x},\ \tau,\ \Delta')$ and by Lemma A.2 we know that $\Gamma \vdash \rho' \ : \ K'$. So by induction hypothesis $\Gamma; K' \vdash e'_2 : \tau \triangleright H'_2; \Lambda'_2$ with $C, H \to^* C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H'$. The judgement $\Gamma; K \vdash e'_s \ : \ \tau \triangleright H'_2; \Lambda'_2$ follows by applying the rule (TDLET) and since $\Lambda'_s = \Lambda'_2 \sqsubseteq \Lambda_2 = \Lambda_s$.
  * rule (DLET2)
    We know that $e'_s = v$ and $\rho \vdash C,\ e_s \to C,\ e'_s$. By hypothesis we know that $\Gamma; K_{\tilde{x}}, (\tilde{x},\ \tau_1,\ \Delta',\ \Lambda') \vdash v \ : \ \tau \triangleright H_2; \bot$ and by the Lemma A.13 we have $\Gamma; K \vdash v \ : \ \tau \triangleright H_2; \bot$ and the thesis follows by choosing vacuously.

- rule (TDVAR)
  By the premise of rule (TDVAR) $K(\tilde{x}) = (\tau,\, \Delta,\, \Lambda)$, where $\Delta = \bigotimes_{i \in \{1,\dots,n\}} ask\, G_i.H_i \otimes \mathit{fail}$, where each H_i abstracts the corresponding expression e_i stored in $\rho(\tilde{x}) = G_1.e_1,\, \dots, G_n.e_n$. We have to prove that $\Gamma; K \vdash e : \tau \triangleright H'; \Lambda$, if $\rho \vdash C,\, \tilde{x} \to C,\, e$, that implies that there exists a $j \in \{1, \dots, n\}$ such that $e = e_j$. Since $\Gamma \vdash \rho : K$ we have that for all $i \in \{1, \dots, n\}$ it holds $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i} \vdash e_i : \tau \triangleright H_i; \Lambda$ where $\gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ and by Lemma A.9 we conclude that $\forall i \in \{1, \dots, n\}$ it is $\Gamma; K \vdash e_i\{\overrightarrow{t_i}/\overrightarrow{y_i}\} : \tau \triangleright H_i; \Lambda$. The thesis holds from Lemma A.14 and $\Lambda \sqsubseteq \Lambda$.
- rule (TAPP)
  By the premise of rule (TAPP) we know that $\Gamma; K \vdash e_1 : \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2 \triangleright H_1; \Lambda_1$, $\Gamma; K \vdash e_2 : \tau_1 \triangleright H_2; \Lambda_2$ and $K' \sqsubseteq K$ hold. Three rules only may drive $\rho \vdash C,\, e_s \to C',\, e'_s$.

  * rule (APP1)
    We know that $e'_s = e'_1\, e_2$. By using the induction hypothesis we have that $\Gamma; K \vdash e'_1 : \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2 \triangleright H'_1; \Lambda'_1$ with $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$ and $\Lambda'_1 \sqsubseteq \Lambda_1$. By the (TAPP) rule we have $\Gamma; K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2 \cdot H_3; \Lambda'_1 \uplus \Lambda_2 \uplus \Lambda_3$ and by Lemma A.14 and Property A.17 we can establish the thesis.
  * rule (APP2)
    We know that $e'_s = (\lambda_f x.e)\, e_2$. By using Lemma A.11 we have $\Gamma; K \vdash \lambda_f x.e : \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2 \triangleright \epsilon; \bot$ and by the induction hypothesis $\Gamma; K \vdash e'_2 : \tau_1 \triangleright H'_2; \Lambda'_2$ with $C,\, H_2 \to^* C',\, \overline{H}$ for some $\overline{H} \sqsubseteq H'_2$ and $\Lambda'_2 \sqsubseteq \Lambda_2$. By (TAPP) $\Gamma; K \vdash e'_s : \tau_2 \triangleright \epsilon \cdot H'_2 \cdot H_3; \bot \uplus \Lambda'_2 \uplus \Lambda_3$ holds. By Property A.15 $\epsilon \cdot H'_2 \cdot H_3 = H'_2 \cdot H_3$ holds, and we also have that $H_1 = (\epsilon + \sum_{i=1}^{1} H_i)$ by Lemma A.12. So $C, H_1 \cdot H_2 \cdot H_3 \to C, \epsilon \cdot H_2 \cdot H_3 \to C, H_2 \cdot H_3 \to^* \overline{H} \cdot H_3$ and the thesis follows by Property A.17 and because $\bot \uplus \Lambda'_2 \uplus \Lambda_3 \sqsubseteq \Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3$.
  * rule (APP3)
    We know that $e'_s = e\{v/x,\, (\lambda_f x.e)/f\}$ and $\rho \vdash C,\, e_s \to C,\, e'_s$. We prove that $\Gamma; K \vdash e\{v/x,\, (\lambda_f x.e)/f\} : \tau_2 \triangleright H_3; \Lambda_3$. By Lemma A.11 we know that $\Gamma; K \vdash e_1 : \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2 \triangleright \epsilon; \bot$ and $\Gamma; K \vdash e_2 : \tau_1 \triangleright \epsilon; \bot$. By hypothesis and Lemma A.8 we conclude that $\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H_3;\Lambda_3} \tau_2; K \vdash e : \tau_2 \triangleright H_3; \Lambda_3$. By Lemma A.9 we have that $\Gamma; K \vdash e\{v/x,\, (\lambda_f x.e)/f\} : \tau_2 \triangleright H_3; \Lambda_3$. The thesis follows because it holds $H_j = (\epsilon + \sum_{i=1}^{n} H_i)$ for $j \in \{1, 2\}$ by Lemma A.12 and because $C, H_1 \cdot H_2 \cdot H_3 \to C, \epsilon \cdot H_2 \cdot H_3 \to C, H_2 \cdot H_3 \to C, \epsilon \cdot H_3 \to C, H_3$.

- rule (TLET)
  By the premise of rule (TLET) we know that $\Gamma; K \vdash e_1 : \tau_1 \triangleright H_1; \Lambda_1$ and $\Gamma, x : \tau_1; K \vdash e_2 : \tau_2 \triangleright H_2; \Lambda_2$ hold. There are only two rules by which $\rho \vdash C,\, e_s \to C',\, e'_s$ can be derived.

  * rule (LET1)
    We know that $e'_s = \mathtt{let}\, x = e'_1\, \mathtt{in}\, e_2$. By the induction hypothesis we have that $\Gamma; K \vdash e'_1 : \tau_1 \triangleright H'_1; \Lambda'_1$ and $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$, $\Lambda'_1 \sqsubseteq \Lambda_1$ and $\Gamma; K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2; \Lambda'_1 \uplus \Lambda_2$. The thesis follows by Lemma A.14 and Property A.17.
  * rule (LET2)
    We know that $e'_s = e_2\{v/x\}$, $\rho \vdash C,\, e_s \to C,\, e'_s$, $\Gamma; K \vdash v : \tau_1 \triangleright H_1; \Lambda_1$ and $\Gamma, x : \tau_1; K \vdash e_2 : \tau_2 \triangleright H_2; \Lambda_2$. By Lemma A.11 $\Gamma; K \vdash v : \tau_1 \triangleright \epsilon; \bot$ and by Lemma A.9 $\Gamma; K \vdash e_2\{v/x\} : \tau_2 \triangleright H_2; \Lambda_2$. By Lemma A.12 we have $H_1 = (\epsilon + \sum_{i=1}^{n} H_i)$, so $C, H_1 \cdot H_2 \to C, \epsilon \cdot H_2 \to C, H_2$ proving the thesis.

- rule (TIF)
  By the premise of rule (TIF) we know that $\Gamma; K \vdash e_1 \ : \ bool \triangleright H_1; \Lambda_1$, $\Gamma; K \vdash e_2 \ : \ \tau \triangleright H_2; \Lambda_2$ and $\Gamma; K \vdash e_3 \ : \ \tau \triangleright H_3; \Lambda_3$ hold. There are three rules only by which $\rho \vdash C,\ e_s \rightarrow C',\ e'_s$ can be derived.

  - rule (IF1)
    We know that $e'_s = \texttt{if}\ e'_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. By using the induction hypothesis we have that $\Gamma; K \vdash e'_1 \ : \ bool \triangleright H'_1; \Lambda'_1$ with $C,\ H_1 \rightarrow^* C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$ and $\Lambda'_1 \sqsubseteq \Lambda_1$. So by rule (TIF) we conclude that $\Gamma; K \vdash e'_s \ : \ \tau \triangleright H'_1 \cdot (H_2 + H_3); \Lambda'_1 \uplus \Lambda_2 \uplus \Lambda_3$ and the thesis follows by Lemma A.14 and Property A.17.
  - rule (IF2)
    We know that $e'_s = e_2$, $\rho \vdash C,\ e_s \rightarrow C,\ e'_s$ and $\Gamma; K \vdash e_2 \ : \ \tau \triangleright H_2; \Lambda_2$. By Lemma A.12 we know $H_1 = (\epsilon + \sum_{i=1}^{n} H_i)$, so the thesis is immediate because $C, H_1 \cdot (H_2 + H_3) \rightarrow C, \epsilon \cdot (H_2 + H_3) \rightarrow C, H_2$.
  - rule (IF3)
    Similar to rule (IF2)

- rule (TSUB)
  By the premise of rule (TSUB) we know $\Gamma; K \vdash e_s \ : \ \tau' \triangleright H'; \Lambda'$, $\tau' \leq \tau$ and $H_s = H' + \overline{H}$ and $\Lambda' \sqsubseteq \Lambda_s$. Then by the induction hypothesis $\Gamma; K \vdash e'_s \ : \ \tau' \triangleright H'_1; \Lambda'_1$, and $C,\ H' \rightarrow^* C',\ H'_2$ for some $H'_2 \sqsubseteq H'_1$ and $\Lambda'_1 \sqsubseteq \Lambda'$. By applying the (TSUB) rule with $H'_s = H'_1 + \overline{H}$ and $\Lambda'_s = \Lambda'$ we have $\Gamma; K \vdash e'_s \ : \ \tau' \triangleright H'_s; \Lambda'_s$ for some Λ'_s. The thesis follows because $C, H_s = C, H' + \overline{H} \rightarrow C, H' \rightarrow^* C', H'_2$ and because $H'_2 \sqsubseteq H'_1 \sqsubseteq H'_s$.
- rule (TFRAME)
  By the premise of the rule TFRAME we know that $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$ holds. There are two rules only by which $\rho \vdash C, e_s \rightarrow C', e'_s$ can be derived:

  - rule (FRAME1)
    We know that $e'_s = \psi^l[e']$, $\rho \vdash C, e_s \rightarrow C', e'_s$. By the induction hypothesis we have that $\Gamma; K \vdash e' \ : \ \tau \triangleright H'; \Lambda'$ and $C, H \ \rightarrow \ C', H''$ for some $H'' \sqsubseteq H'$ and $\Lambda' \sqsubseteq \Lambda$. By applying the rule (TFRAME) we have $\Gamma; K \vdash e'_s \ : \ \tau \triangleright \psi^{l'}[H']; \Lambda' \uplus [l' \mapsto l]$ and $C, \psi^{l'}[H] \rightarrow C', \psi^{l'}[H'']$ (by Property A.17) and $\Lambda' \uplus [l' \mapsto l] \sqsubseteq \Lambda \uplus [l' \mapsto l]$.
  - rule (FRAME2)
    We know that $e'_s = v$ and $\rho \vdash C, e_s \rightarrow C, e'_s$. By hypothesis we know that $\Gamma; K \vdash v : \tau \triangleright H; \Lambda$ and by Lemma A.11 we have $\Gamma; K \vdash v : \tau \triangleright \epsilon; \bot$ so the thesis follows immediately.

□

Corollary 1 (Over-approximation). *Let e be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e and $\Gamma \vdash \rho \ : \ K$.*
If $\Gamma; K \vdash e \ : \ \tau \triangleright H; \Lambda$ and $\rho \vdash C,\ e \rightarrow^* C',\ e'$, then $\Gamma; K \vdash e' \ : \ \tau \triangleright H'; \Lambda'$ and there exists a sequence of transitions $C,\ H \rightarrow^* C',\ H''$, for some $H'' \sqsubseteq H'$ and $\Lambda' \sqsubseteq \Lambda$.

Proof. An easy inductive reasoning on the length of the computation suffices to prove the statement by using Theorem 4.1. □

Theorem 4.2 (Progress).
Let e be a closed expression such that $\Gamma; K \vdash e \ : \ \tau \triangleright H; \Lambda$; and let ρ be a dynamic environment such

that $dom(\rho)$ includes the set of parameters of e, and $\Gamma \vdash \rho \,:\, K$.
If $\rho \vdash C,\ e \not\rightarrow$; e violates no policies; and H is viable for C (i.e. $C,\ H \not\rightarrow^{+} C', fail$), then e is a value.

Proof. Let below $DFV(e)$ be the set of the free dynamic variables occurring in e. By induction on the depth of the typing derivations and then by cases on the last rule applied. The cases (TCONST), (TFACT), (TABS), (TVARIATION) are immediate since e_s is a value. The case (TVAR) cannot occur because e_s is closed with respect to identifiers. So we assume that e_s is not a value and it is stuck in C.

- rule (TIF) $\qquad e_s = \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3$
  If e_s is stuck, then it is only the case that e_1 is stuck. By induction hypothesis this can occur only when e_1 is a value. Since $\Gamma; K \vdash e_1 \,:\, bool \triangleright H_1; \Lambda_1$ by our hypothesis and $v = true$ or $v = false$ by Lemma A.7(1), either rule (IF2) or (IF3) applies, contradiction.
- rule (TLET) $\qquad e_s = \mathtt{let}\, x \,=\, e_1\, \mathtt{in}\, e_2$
  If e_s is stuck, then it is only the case that e_1 is stuck. By induction hypothesis this can occur only when e_1 is a value, hence (LET2) rule applies, contradiction.
- rule (TTELL) $\qquad e_s = \mathtt{tell}(e)$
  If e_s is stuck, then it is only the case that e is stuck. By induction hypothesis this can occur only when e is a value v and by Lemma A.7(4) $v = F$, so the rule (TELL2) applies, contradiction.
- rule (TRETRACT) $\qquad e_s = \mathtt{retract}(e)$
  Similar to the (TTELL) case.
- rule (TAPPEND) $\qquad e_s = e_1 \cup e_2$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(3) $e_1 = (x)\{Va\}$. If e_2 reduces, rule (VAPPEND2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value and Lemma A.7(3) $e_2 = (y)\{Va\}$, hence, rule (VAPPEND3) applies, contradiction.
- rule (TSUB)
  Straightforward by induction hypothesis
- rule (TAPP) $\qquad e_s = e_1\, e_2$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(2) $e_1 = \lambda_f x.e$. If e_2 reduces, rule (APP2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value, hence, rule (APP3) applies, contradiction.
- rule (TDVAR) $\qquad e_s = \tilde{x}$
  If e_s is stuck we can have two cases only. The first case is that $\tilde{x} \notin dom(\rho)$. But this is not possible because $DFV(e_s) \subseteq dom(\rho)$ by our hypothesis. The second case is that $\rho(\tilde{x}) = Va$ and $dsp(C, Va)$ is not defined. But this is not possible because $C,\ H_s \not\rightarrow^{\star} C', fail$ by our hypothesis, so the $dsp(C, Va)$ is defined and (DVAR) rule applies, contradiction.
- rule (TVAPP) $\qquad e_s = \#(e_1, e_2)$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(3) $e_1 = (x)\{Va\}$. If e_2 reduces, rule (VAAPP2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value, hence, rule (VAAPP3) applies, contradiction.
- rule (TDLET) $\qquad e_s = \mathtt{dlet}\, \tilde{x} \,=\, e_1\, \mathtt{when}\, G\, \mathtt{in}\, e_2$
  If e_s is stuck, then it is only the case that e_2 is stuck. By the premise of (TDLET) rule and by Lemma 4.2 $\Gamma \vdash \rho' \,:\, K'$ with $\rho' = \rho[G.e, \rho(\tilde{x})/\tilde{x}]$ and $K' = K_{\tilde{x}}, (\tilde{x}, \tau, \Delta', \Lambda')$. Since $DFV(e_2) \subseteq$

$DFV(e_s) \subseteq dom(\rho) \subseteq dom(\rho')$ we can apply the induction hypothesis so e_2 is a value. In this case the (DLET2) rule applies, contradiction.

- rule (TFRAME) $\quad e_s = \psi[e]$ If e_s is stuck, then it is only the case that e is stuck because we assumed that e_s violates no policies. But the induction hypothesis says that e is a value, thus the rule (FRAME2) can be applied (contradiction).

Theorem 4.3 (Correctness)**.**
Let e_s be a closed expression such that $\Gamma; K \vdash e_s \ : \ \tau \triangleright H_s; \Lambda_s$; let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and that $\Gamma \vdash \rho \ : \ K$; finally let C be a context such that $C, H_s \not\twoheadrightarrow^{+} C', fail$. Then either the computation of e_s terminates yielding a value ($\rho \vdash C, e_s \rightarrow^{*} C'', v$) or it diverges, but it never gets stuck.

(By contradiction). Assume that for some i it is $\rho \vdash C, e_s \rightarrow^{i} C'', e_s^i \not\twoheadrightarrow$ where e_s^i is a non-value stuck expression. By Proposition 1 we have $\Gamma; K \vdash e_s^i \ : \ \tau \triangleright H_s^i; \Lambda$ and $C, H_s \rightarrow^{\star} C'', H_s^i$, and since $C, H_s \not\twoheadrightarrow C', fail$ we have also $C, H_s^i \not\twoheadrightarrow C', fail$. Then, Theorem 4.2 suffices to show that e_s^i is a value (contradiction). □

B. Properties of the Load Time Analysis

In this subsection we prove some properties about the load time analysis in Section 5, in particular we prove Theorems 5.1, 5.2. Firstly, we define the complete lattice of the analysis estimate by ordering $\wp(Context)$ and $\wp(Policies)$ by inclusion and by exploiting the standard construction of cartesian product and functional space. More in detail, we write $(\Sigma^1_\circ, \Sigma^1_\bullet, \Psi^1_\circ, \Psi^1_\bullet) \sqsubseteq (\Sigma^2_\circ, \Sigma^2_\bullet, \Psi^2_\circ, \Psi^2_\bullet)$, whenever their components are ordered by pointwise set inclusion. Furthermore, their meet $(\Sigma^1_\circ, \Sigma^1_\bullet, \Psi^1_\circ, \Psi^1_\bullet) \sqcap (\Sigma^2_\circ, \Sigma^2_\bullet, \Psi^2_\circ, \Psi^2_\bullet) = (\Sigma^1_\circ \sqcap \Sigma^2_\circ, \Sigma^1_\bullet \sqcap \Sigma^2_\bullet, \Psi^1_\circ \sqcap \Psi^2_\circ, \Psi^1_\bullet \sqcap \Psi^2_\bullet)$, where $\sqcap$ is pointwise set intersection.

By exploiting standard lattice theory it is straightforward to prove that analysis estimates are a complete lattice. In the following we denote the top of this lattice with $(\Sigma^\top, \Sigma^\top, \Psi^\top, \Psi^\top)$.

Lemma B.1.

1. *For all H^l $(\Sigma^\top, \Sigma^\top, \Psi^\top, \Psi^\top) \models H^l$*
2. *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be such that $\mathcal{E}_1 \models H^l$ and $\mathcal{E}_2 \models H^l$ then $\mathcal{E}_1 \sqcap \mathcal{E}_2 \models H^l$*

Proof. The thesis is proved by induction on the structure of H^l and by using the analysis rules. The proof is quite standard and below we only discuss the case $H^l = (H_1^{l_1} \cdot H_2^{l_2})^l$.

1. By induction hypothesis $(\Sigma^\top, \Sigma^\top, \Psi^\top, \Psi^\top) \models H_i^{l_i}$ for $i \in \{1, 2\}$. By definition $\forall l' \in Lab$ $\Sigma^\top(l') = Context$ and $\Psi^\top(l') = Policies$. Then, it holds $\Sigma^\top(l) \subseteq \Sigma^\top(l_1)$, $\Sigma^\top(l_1) \subseteq \Sigma^\top(l_2)$ and $\Sigma^\top(l_2) \subseteq \Sigma^\top(l)$. Also, it holds $\Psi^\top(l) \subseteq \Psi^\top(l_1)$, $\Psi^\top(l_1) \subseteq \Psi^\top(l_2)$ and $\Psi^\top(l_2) \subseteq \Psi^\top(l)$. These inclusions satisfy the premises of the rule (ASEQ1), then we conclude $(\Sigma^\top, \Sigma^\top, \Psi^\top, \Psi^\top) \models H^l$.
2. From hypothesis for $i = 1, 2$ $\mathcal{E}_i = (\Sigma^i_\circ, \Sigma^i_\bullet, \Psi^i_\circ, \Psi^i_\bullet) \models H^l$ and from the premises of rule (ASEQ1), we know that $\Sigma^i_\circ(l) \subseteq \Sigma^i_\circ(l_1)$, $\Sigma^i_\bullet(l_1) \subseteq \Sigma^i_\circ(l_2)$, $\Sigma^i_\bullet(l_2) \subseteq \Sigma^i_\bullet(l)$ and that $\Psi^i_\circ(l) \subseteq \Psi^i_\circ(l_1)$, $\Psi^i_\bullet(l_1) \subseteq \Psi^i_\circ(l_2)$, $\Psi^i_\bullet(l_2) \subseteq \Psi^i_\bullet(l)$. Since $\cap$ is monotonic with respect to $\subseteq$ it holds $\Sigma^1_\circ(l) \cap \Sigma^2_\circ(l) \subseteq \Sigma^1_\circ(l_1) \cap \Sigma^2_\circ(l_1)$, $\Sigma^1_\bullet(l_1) \cap \Sigma^2_\bullet(l_1) \subseteq \Sigma^1_\circ(l_2) \cap \Sigma^2_\circ(l_2)$, $\Sigma^1_\bullet(l_2) \cap \Sigma^2_\bullet(l_2) \subseteq \Sigma^1_\bullet(l) \cap \subseteq \Sigma^2_\bullet(l)$. The same holds for the policies, $\Psi^1_\circ(l) \cap \Psi^2_\circ(l) \subseteq \Psi^1_\circ(l_1) \cap \Psi^2_\circ(l_1)$, $\Psi^1_\bullet(l_1) \cap \Psi^2_\bullet(l_1) \subseteq \Psi^1_\circ(l_2) \cap \Psi^2_\circ(l_2)$, $\Psi^1_\bullet(l_2) \cap \Psi^2_\bullet(l_2) \subseteq \Psi^1_\bullet(l) \cap \subseteq \Psi^2_\bullet(l)$. Then the induction hypothesis and the above inclusions fulfil the premises of rule (ASEQ1) and we conclude $(\Sigma^1_\circ \sqcap \Sigma^2_\circ, \Sigma^1_\bullet \sqcap \Sigma^2_\bullet) \models H^l$. □

By exploiting the above two lemmata we can prove

Theorem 5.1 (Existence of estimates).
Given H^l and an initial context C, the set $\{\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \mid \mathcal{E} \vDash H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.

Proof. We need to show that given a set of solutions $Y = \{(\Sigma^i_\circ, \Sigma^i_\bullet, \Psi^i_\circ, \Psi^i_\bullet) \mid i \in \{1, \ldots, n\}\} \subseteq X = \{(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H^l\}$, it is $\sqcap Y \in X$. By applying $n+1$ times Lemma B.1 we have that $(\Sigma^\top, \Sigma^\top, \Psi^\top, \Psi^\top) \sqcap (\Sigma^1_\circ, \Sigma^1_\bullet, \Psi^1_\circ, \Psi^1_\bullet) \sqcap \cdots \sqcap (\Sigma^n_\circ, \Sigma^n_\bullet, \Psi^n_\circ, \Psi^n_\bullet) \vDash H^l$ holds. □

Two prove the subject reduction we need the following definition and two lemmata.

Definition B.1 (Immediate subterm). Let H and H_1 be history expressions (for simplicity we omit labels). Call H_1 immediate subterm of H if H is of either form $H_1 + H_2$ or $H_2 + H_1$ or $H_1 \cdot H_2$ or $H_2 \cdot H_1$ or $\mu h.H_1$ or $askG.H_1 \otimes \Delta$ or $\psi[H]$.

Lemma B.2 (Pre-substitution). *Let H^l, $H_1^{l_1}$ and $H_2^{l_2}$ be history expressions such that $H_1^{l_1}$ is an immediate subterm of H^l; let $\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ be such that $\mathcal{E} \vDash H^l$, $\mathcal{E} \vDash H_1^{l_1}$ and $\mathcal{E} \vDash H_2^{l_2}$.*
If $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1)$, $\Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$, $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1)$ then $\mathcal{E} \vDash H^l[H_2^{l_2}/H_1^{l_1}]$.

Proof. The proof is by cases on the structure of H^l.

- case $\ni$, ϵ^l, $tell\,F^l$, $fail^l$, $retract\,F^l$, h^l
  straightforward.
- case $H^l = (H_1^{l_1} + H_3^{l_3})^l$
  From the hypothesis and from the premises of rule (ASUM) it holds $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$, $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$. So by applying the rule (ASUM) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash (H_2^{l_2} + H_3^{l_3})^l$.
- case $H^l = (H_3^{l_3} + H_1^{l_1})^l$
  Similar to the previous case.
- case $H^l = (H_1^{l_1} \cdot H_3^{l_3})^l$
  From the hypothesis and the premises of rule (ASEQ1) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l_3)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$ and $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \subseteq \Psi_\bullet(l_3)$. So by applying the rule (ASEQ1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash (H_2^{l_2} \cdot H_3^{l_3})^l$.
- case $H^l = (H_3^{l_3} \cdot H_1^{l_1})^l$
  From the hypothesis and the premises of rule (ASEQ1) we have $\Sigma_\bullet(l_3) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\bullet(l_3) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$ and $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$. So by applying the rule (ASEQ1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash (H_3^{l_3} \cdot H_1^{l_1})^l$.
- case $H^l = (\mu h.H_1^{l_1})^l$
  From the hypothesis and the premises of rule (AREC) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$ and $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$. So by applying the rule (AREC) with the new inclusions we have $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash (\mu h.H_2^{l_2})^l$.
- case $H^l = (askG.H_1^{l_1} \otimes \Delta^{l_3})^l$
  From the hypothesis and the premises of rule (AASK1) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$, $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$. So by the rule (AASK1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash (askG.H_2^{l_2} \otimes \Delta^{l_3})^l$.

- case $H^l = \psi^l[H_1^{l_1}]$
  From the hypothesis and the premises of rule (AASK1) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\{\psi\} \subseteq \Psi_\circ(l) \subseteq \Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$, $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1) \backslash \{\psi\} \subseteq \Psi_\bullet(l)$. So by the rule (AFRAME) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models \psi^l[H_2^{l_2}]$. □

Lemma B.3 (Substitution). *Let H^l, $H_1^{l_1}$ and $H_2^{l_2}$ be history expressions such that $H_1^{l_1}$ is a subterm of H^l; let $\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet)$ be such that $\mathcal{E} \models H^l$, $\mathcal{E} \models H_1^{l_1}$ and $\mathcal{E} \models H_2^{l_2}$.*
If $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1)$, $\Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$, $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1)$ then $\mathcal{E} \models H^l[H_2^{l_2}/H_1^{l_1}]$.

Proof. Since $H_1^{l_1}$ is subterm of H^l, there exists then another subterm of H^l, say $H_3^{l_3}$, such that $H_1^{l_1}$ is an immediate subterm of $H_3^{l_3}$. Since $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H^l$, there exists then a subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_3^{l_3}$. Since $H_1^{l_1}$ is an immediate subterm of $H_3^{l_3}$ there exists another subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_1^{l_1}$. So by applying Lemma B.2, we have $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_3^{l_3}[H_2^{l_2}/H_1^{l_1}]$. Since our analysis is defined on the syntax of history expressions and since $\Sigma_\circ(l_3)$, $\Sigma_\bullet(l_3)$, $\Psi_\circ(l_3)$ and $\Psi_\bullet(l_3)$ have not changed, we can reuse the same steps used for $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H^l$ to prove $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H^l[H_2^{l_2}/H_1^{l_1}]$. □

Theorem 5.2 (Subject Reduction).
Let H_1^l be a closed history expression and let $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_1^l$. If for all $C \in \Sigma_\circ(l)$ we have that $C, H_1^l \to C', H_2^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_2^{l'}$; $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$; $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$; $\Psi_\circ(l) \subseteq \Psi_\circ(l')$; and $\Psi_\bullet(l') \subseteq \Psi_\bullet(l)$.

Proof. By induction on the depth of the analysis derivation and then by cases on the last rule applied.

- rule (ANIL)
  The statement holds vacuously.
- rule (AASK2)
  The statement holds vacuously.
- rule (AEPS)
  We know that in this case $C, \epsilon^l \to C$, э, then the statement holds vacuously.
- rule (ATELL)
  We know that in this case $C, \mathit{tell}\, F^l \to C \cup \{F\}$, э, then the statement holds vacuously.
- rule (ARETRACT)
  Similar to (ATELL) rule
- rule (ASEQ1)
  In this case we have $H = (H_1^{l_1} \cdot H_2^{l_2})^l$ and $H' = (H_3^{l_3} \cdot H_2^{l_2})^l$. We have to prove $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models (H_3^{l_3} \cdot H_2^{l_2})^l$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l)$ (trivial), $\Sigma_\bullet(l) \subseteq \Sigma_\bullet(l)$ (trivial), $\Psi_\circ(l) \subseteq \Psi_\circ(l)$ (trivial) and $\Psi_\bullet(l) \subseteq \Psi_\bullet(l)$ (trivial). The premises of (ASEQ1) guarantee that $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_1^{l_1}$, $(\Sigma_\circ, \Sigma_\bullet) \models H_2^{l_2}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\circ(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1)$, $\Psi_\bullet(l_1) \subseteq \Psi_\circ(l_2)$ and $\Psi_\bullet(l_2) \subseteq \Psi_\circ(l)$. By the premises of the semantic rule it holds $C, H_1^{l_1} \to C', H_3^{l_3}$. The induction hypothesis says that $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models H_3^{l_3}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_3)$, $\Sigma_\bullet(l_3) \subseteq \Sigma_\bullet(l_1)$, $\Psi_\circ(l_1) \subseteq \Psi_\circ(l_3)$ and $\Psi_\bullet(l_3) \subseteq \Psi_\bullet(l_1)$. So transitivity of set inclusion suffices to fulfill the premises of (ASEQ1) and to conclude that $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \models (H_3^{l_3} \cdot H_2^{l_2})^l$ holds.

- rule (ASEQ2)
  In this case we know $H^l = (\backepsilon \cdot H_2^{l_2})^l$ and $H'^{l'} = H_2^{l_2}$. The thesis is straightforward by the premises of (ASEQ2) rule.
- rule (ASUM)
  In this case we have $H^l = (H_1^{l_1} + H_2^{l_2})^l$ and two cases for $H'^{l'}$:

  1. case $H'^{l'} = H_1'^{l'_1}$. By the semantic rule we know $C, H_1^{l_1} \to C', H_1'^{l'_1}$, and by the induction hypothesis $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_1'^{l'_1}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l'_1)$ and $\Sigma_\bullet(l'_1) \subseteq \Sigma_\bullet(l_1)$, $\Psi_\circ(l_1) \subseteq \Psi_\circ(l'_1)$, $\Psi_\bullet(l'_1) \subseteq \Psi_\bullet(l_1)$. The thesis follows then by transitivity of set inclusion.
  2. case $H'^{l'} = H_2'^{l'_2}$. Similar to case (1).

- rule (AASK1)
  In this case we have $H^l = (askG.\, H_1^{l_1} \otimes \Delta^{l_2})^l$ and two cases for $H'^{l'}$:

  1. case $H'^{l'} = H_1^{l_1}$. In this case we know $C \vDash G$ and the thesis follow immediately from the premises of (AASK1).
  2. case $H'^{l'} = \Delta^{l_2}$. Similar to case (1) with the only difference that $C \nvDash G$.

- rule (AREC)
  In this case we know $H^l = (\mu.H_1^{l_1})^l$ and $H'^{l'} = H_1^{l_1}[(\mu.H_1^{l_1})^l/h]$. The premises of the rule guarantee that $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_1^{l_1}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1)$ and $\Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$. We have two cases

  1. h does not occur in H_1. In this case the thesis trivially follows since $H_1^{l_1}[(\mu.H_1^{l_1})^l/h] = H_1$.
  2. h occurs n times with labels $l^1, \dots, l^n$. Since our analysis is defined on the syntax of history expressions, in the proof of $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_1^{l_1}$ there exists a subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash h^{l^i}$. By the premises of the rule (AVAR) we know that $\Sigma_\circ(l_i) \subseteq \Sigma_\circ(l)$, $\Sigma_\bullet(l) \subseteq \Sigma_\bullet(l_i)$, $\Psi_\circ(l_i) \subseteq \Psi_\circ(l)$ and $\Psi_\bullet(l) \subseteq \Psi_\bullet(l_i)$. So by applying Lemma B.3 n-times, we have $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_1^{l_1}[(\mu.H_1^{l_1})^l/h^{l^i}]$ for $i \in \{1, \dots, n\}$. Now the premises of rule (AREC) suffices to establish that $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Psi_\circ(l) \subseteq \Psi_\circ(l_1)$ and $\Psi_\bullet(l_1) \subseteq \Psi_\bullet(l)$.

- rule (AFRAME)
  In this case we have $H^l = \psi^l[H_1^{l_1}]$ and $C, H_1^{l_1} \to C', H_2^{l_2}$. By the premises of the rule (AFRAME) we know $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_1^{l_1}$, and by induction hypothesis it holds $(\Sigma_\circ, \Sigma_\bullet, \Psi_\circ, \Psi_\bullet) \vDash H_2^{l_2}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1)$, $\Psi_\circ(l_1) \subseteq \Psi_\circ(l_2)$ and $\Psi_\bullet(l_2) \subseteq \Psi_\bullet(l_1)$. The thesis follows by transitivity of set inclusion and by applying the rule (AFRAME).

□