Linguistic Mechanisms for Context-Aware Security [⋆]

Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori

Dipartimento di Informatica, Università di Pisa
{chiara, degano, galletta}@di.unipi.it, francesco.salvatori@sns.it

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 extend (the core of) an adaptive functional language with primitives to enforce security policies on the code execution, and we exploit a static analysis to instrument programs. The introduced checks guarantee that no violation of the required security policies occurs.

1 Introduction

Context and Adaptivity. 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 or their non-functional requirements, e.g. security or quality of service. Therefore, languages need effective mechanisms to sense the changes in the operational environment, i.e. the *context*, in which the application is plugged in, and to properly *adapt* to changes. At the same time, these mechanisms must maintain the functional and non-functional properties of applications after the adaptation steps.

The context is a key notion for adaptive software. Typically, a context includes different kinds of computationally accessible information coming both from 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.).

Context Oriented Programming (COP) [9,15,1,17,3] is a recent paradigm that proposes linguistic features to deal with contexts and adaptivity. Its main construct is *behavioural variation*, a chunk of code to be activated depending on the current context hosting the application, to dynamically modify the execution.

Security and Contexts. The combination of security and context-awareness requires to address two aspects. First, security may reduce adaptivity, by adding further constraints on the possible actions of software. Second, new highly dynamic security mechanisms are needed to scale up to adaptive software. In the literature, e.g. in [26,6], this duality is addressed in two ways: *securing context-aware systems* and *context-aware security*.

Securing context-aware systems aims at rephrasing the standard notions and techniques for confidentiality, integrity and availability [24], and at developing techniques for guaranteeing them [26]. The challenge is to understand how to get secure and trusted context information. Context-aware security is dually concerned with the use of context

[⋆] Work partially supported by the MIUR-PRIN project Security Horizons.

information to dynamically drive security decisions. Consider the usual no flash photography policy in museums. While a standard security policy *never* allows people to use flash, a context-aware security could forbid flash *only* inside particular rooms.

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 [25] or in the interaction protocols [14], that mostly address access control for resources and for smart things (see e.g. [26,16,27], and [2,10]). More foundational issues have been less studied within the programming languages approach we follow; preliminary work can be found, e.g. in [7,23].

Our proposal. The kernel of our proposal relies on extending $\mathrm{ML}_{\mathrm{CoDa}}$, a core ML with COP features introduced in [13]. Its main novelty is to be a two-component language: a declarative part for programming the context and a functional one for computing.

The context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented as a Datalog program (stratified, with negation) [22,19]. 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; context is changed by using the standard $tell/retract$ constructs.

Programming adaptation is specified through behavioural variations, a first class, higher-order $\mathrm{ML}_{\mathrm{CoDa}}$ construct. They can be referred to by identifiers, and used as a parameter in functions. This fosters dynamic, compositional adaptation patterns, as well as reusable, modular code. The chunk of a behavioural variation to be run is selected by the *dispatching* mechanism that inspects the actual context and makes the right choices.

Notably, $\mathrm{ML}_{\mathrm{CoDa}}$, as it is, offers the features needed for addressing context-aware security issues, in particular for defining and enforcing access control policies. Our version of Datalog is powerful enough to express all relational algebras, is fully decidable, and guarantees polynomial response time [12]. Furthermore, adopting a stratified-negation-model is common and many logical languages for defining access control policies compile in Stratified Datalog, e.g. [5,18,11]. Here, we are only interested in policies imposed by the system, which are unknown at development time. Indeed the policies of the application can be directly encoded by the developer as behavioural variations. The dispatching mechanism then suffices for checking whether a specific policy holds, and for enabling the chunk of behaviour that obeys it. Our language therefore requires no extensions to deal with security policies.

Our aim is to handle, as soon as possible, both failures in adaptation to the current context (*functional failure*) and policy violations (*non-functional failure*). Note that the actual value of some elements in the current context is only known when the application is linked with it at runtime. Actually, we have a sort of *runtime monitor*, natively supported by the dispatching mechanism of $\mathrm{ML}_{\mathrm{CoDa}}$, which we switch on and off at need. To specify and implement the runtime monitor, we conservatively extend the two-phase verification of [13]. The first phase is based on a type and effect system that, at compile time, computes a safe over-approximation, call it H, of the application behaviour. Then H is used at loading time to verify that (i) the resources required by the application are available in the actual context, and in its future modifications (as done in [13]); and (ii) to detect within the application where a policy violation may occur, i.e. when the context is modified through $tell$ and $retract$ actions.

The loading time analysis requires first to build a graph $\mathcal{G}$, that safely approximates which contexts the application will pass through, while running. While building the graph, we also label its edges with the $tell/retract$ operations in the code, exploiting the approximation H. Before launching the execution, we detect the unsafe operations by checking the policy Φ on each node of $\mathcal{G}$. Our runtime monitor can guard them, where it will be switched off for the remaining actions. Actually, we collect the labels of the risky operations and associate the value `on` with them, and `off` with all the others.

To make the above effective, the compiler instruments the code by substituting a behavioural variation *bv* for each occurrence of a $tell/retract$. At runtime, *bv* checks if Φ holds in the running context, but only when the value of the label is `on`.

The next section introduces $\mathrm{ML}_{\mathrm{CoDa}}$ and our proposal, with the help of a running example, along with an intuitive presentation of the various components of our two-phase static analysis, and the way security is dynamically enforced. The formal definitions and the statements of the correctness of our proposal will follow in the remaining sections. The conclusion summarises our results and discusses some future work.

2 Running example

Consider a multimedia guide to a museum implemented as a smartphone application, starting from the case study in [13]. Assume the museum has a wireless infrastructure exploiting different technologies, like WiFi, Bluetooth, Irda or RFID. When a smartphone is connected, the visitors can access the museum Intranet and its website, from which they download information about the exhibit and further multimedia contents. Each exhibit is equipped with a wireless adapter (Bluetooth, Irda, RFID) and a QR code. They are only used to offer the guide with the URL of the exhibit, retrievable by using one of the above technologies, provided that it is available on the smartphone. If equipped with a Bluetooth adapter, the smartphone connects to that of the exhibit and directly downloads the URL; if the smartphone has a camera and a QR decoder, the guide can retrieve the URL by taking a picture of the code and decoding it.

The smartphone capabilities are stored in the context as Datalog clauses. Consider the following clauses defining when the smartphone can either directly download the URL (the predicate `device(d)` holds when the device `d` $\in$ {`irda`, `bluetooth`, `rfid_reader`} is available), or it can take the URL by decoding a picture (the parameter `x` in the predicate `use_qrcode` is a handle for using the decoder):

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

Contextual data, like the above predicates use_qrcode(decoder) and direct_comm(), affect the download. To change the program flow in accordance to the current context, we exploit behavioural variations. Syntactically, they are similar to pattern matching, where Datalog goals replace patterns and parameters can additionally occur. Behavioural variations are similar to functional abstractions, but their application triggers a *dispatching mechanism* that, at runtime, inspects the context and selects the first expression whose goal holds.

In the following function getExhibitData, we declare the behavioural variation url (with an unused argument "_"), that returns the URL of an exhibit. If the smartphone can directly download the URL, then it does, through the channel returned by the function getChannel(); otherwise the smartphone takes a picture of the QR code and decodes it. In the last case, the variables decoder and cam will be assigned to the handles of the decoder and the one of the camera deduced by the Datalog machinery. These handles are used by the functions take_picture and decode_qr to interact with the actual smartphone resources.

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

The behavioural variation (bound to) url is applied before the getRemoteData call that connects to the corresponding website and downloads the required information (we use here a slightly simplified syntax, for details see Sect. 3).

By applying the function getExhibitData to unit and assuming n is returned by getChannel, we have the following computation ($C, e \rightarrow^{\star} C', e'$ says that the expression e in the context C reduces in several steps to e' changing the context in C'):

$$C, \texttt{getExhibitData}() \rightarrow^{\star} C, \texttt{getRemoteData\#}u \rightarrow^{\star} C, \texttt{getRemoteData}(\texttt{receiveData}\,n)$$

The second configuration above transforms into the third, because C satisfies the goal $\leftarrow$ direct_comm(), and so the dispatching mechanism selects the first expression of the behavioural variation u (the one bound to url in getExhibitData).

To dynamically update the context, we use the constructs *tell* and *retract*, that add and remove Datalog facts. In our example the context stores information about the room in which the user is, through the predicate current_room. If the user moves from the *delicate paintings* room to the *sculptures* one, the application updates the context by:

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

Assume now that one can take pictures in every room, but that in the rooms with delicate paintings it is forbidden to use the flash so to prevent the exhibits from damages. This policy is specified by the museum (the system) and it must be enforced during the user's tour. Since policies predicate on the context, they are easily expressed as Datalog goals. Let the fact `flash_on` hold when the flash is active and the fact `button_clicked` when the user presses the button of the camera. The above policy intuitively corresponds to the logical condition $current\_room(delicate\_paintings) \Rightarrow (button\_clicked \Rightarrow \neg flash\_on)$ and is expressed in Datalog as the equivalent goal

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

Of course, the museum can specify other policies, and we assume that there is a unique global policy Φ (referred in the code as `phi`), obtained by suitably combining them all. The enforcement is obtained by a runtime monitor that checks the validity of Φ right before every context change, i.e. before every *tell*/*retract*.

An application fails to adapt to a context (*functional failure*), when the dispatching mechanism fails. Consider the evaluation of `getExhibitData` on a smartphone without wireless technology and QR decoder. Since no context will ever satisfy the goals of `url`, it gets stuck. Another kind of failure happens when a *tell*/*retract* causes a policy violation (*non-functional failure*). If the context includes `current_room(delica–te_paintings)`, such a violation occurs when attempting to use the flash.

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

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 runtime behaviour of e, called the *history expression*. The effect abstractly represents the changes and the queries performed on the context during its evaluation. Consider the expression:

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

For clarity, we show the labels i of `tell` F_j^i in the code, that are inserted by the compiler while parsing (same for `retract`). Let the facts above be $F_1 \equiv$ `camera_on`; $F_2 \equiv$ `flash_on`; $F_3 \equiv$ `mode_museum_activated`; $F_4 \equiv$ `button_clicked`. The type of e_a is `unit`, as well as that of `tell` F_4, and its history expression is

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

($\cdot$ abstracts sequential composition, $+$ `if-then-else`). Depending upon the value of `always_flash`, that records whether the user wants the flash to be always usable, the expression e_a can either perform the action `tell` F_1, followed by `tell` F_2, *or* the action `tell` F_1, followed by `tell` F_3, so recording in the context that the flash is on or off. After that, e_a will perform `tell` F_4, no matter what the previous choice was.

The labels of history expressions allow us to link the actions in histories to the corresponding actions of the code, e.g. the first `tell` F_1^1 in H_a, corresponds to the first

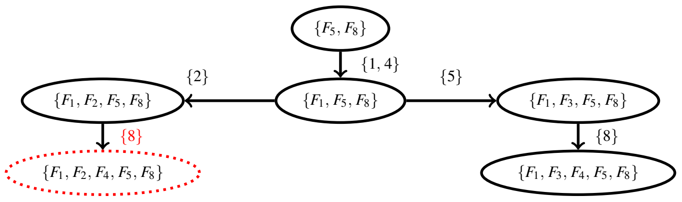

Fig. 1. The evolution graph for the context $\{F_5, F_8\}$ and for the history expression $H_a = (((tell\,F_1^1.tell\,F_2^2)^3 + (tell\,F_1^4.tell\,F_3^5)^6)^7.\underline{tell\,F_4^8})^9$

$\mathtt{tell\,F_1}$ in e_a, that is also labelled by 1, while the $\mathtt{tell\,F_4^8}$ in H_a, is linked to the action with label 5 in e_a. All the correspondences are $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$ (the abstract labels that do not annotate $tell/retract$ actions have no counterpart).

Consider now an initial context C that includes the facts F_5 (irrelevant here), and $F_8 \equiv \mathtt{current\_room(delicate\_paintings)}$, but no facts in $\{F_1, F_2, F_3, F_4\}$. Starting from C (and from H_a) our *loading time* analysis builds the graph described in Fig. 1. Nodes represent contexts, possibly reachable at runtime, while edges represent transitions from one context to another. Each edge is annotated with the set of actions in H_a that may cause that transition, e.g. from the context C it is possible to reach a (single) context that also includes the fact F_1, because of the two *tell* operations labelled by 1 and by 4 in H_a. As a matter of fact, an edge can have an annotation including more than one label (e.g. the one labelled $\{1,4\}$). Note also that the same label may occur in the annotation of more than one edge (e.g. the label 8).

By visiting the graph, we observe that the context $\{F_1, F_2, F_4, F_5, F_8\}$ (the dotted node in Fig. 1, red in the pdf) violates our no-flash policy. At runtime the action labelled with 8 (underlined and in red in the pdf), corresponding to $\mathtt{tell\,F_4}$ must be blocked. For preventing this violation, all we have to do is activate the runtime monitor, right before executing this risky operation.

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

Below, we survey the syntax and the semantics of ML$_{\text{CoDa}}$; for more details see [13].

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. We assume that each program is safe, and we adopt *Stratified Datalog*, under the Closed World Assumption to deal with negation [8]. Security policies are simply expressed as Datalog goals, the value of which is true only if the policy holds.

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.

$$Va ::= G.e \mid G.e, Va \qquad\qquad v ::= c \mid \lambda_f x.e \mid (x)\{Va\} \mid \mathtt{F}$$
$$e ::= v \mid x \mid \tilde{x} \mid e_1\, e_2 \mid let\, x\, =\, e_1\, in\, e_2 \mid if\, e_1\, then\, e_2\, else\, e_3 \mid$$
$$dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2 \mid tell(e_1)^l \mid retract(e_1)^l \mid e_1 \cup e_2 \mid \#(e_1, e_2)$$

To facilitate our static analysis (see Sect. 5) we associate each $tell/retract$ with a label $l \in Lab_C$ (this does not affect the dynamic semantics).

COP-oriented constructs of $\mathrm{ML}_{\mathrm{CoDa}}$ include behavioural variations $(x)\{Va\}$, each consisting of a *variation* Va, i.e. a list of expressions $G_1.e_1, \ldots, G_n.e_n$ guarded by Datalog goals G_i (x free in e_i). At runtime, the first goal G_i satisfied by the context selects the expression e_i to be run (*dispatching*). 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. The $tell/retract$ constructs update the context by asserting/retracting facts, provided that the resulting context satisfies the system policy Φ. The append operator $e_1 \cup e_2$ concatenates behavioural variations, so allowing for dynamically composing them. The evaluation of 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.

Semantics The Datalog component has the standard top-down semantics [8]. 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.

The SOS semantics of $\mathrm{ML}_{\mathrm{CoDa}}$ 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 $\rho \vdash C,\, e \to C',\, e'$ says that in 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.

Fig. 2 shows the inductive definitions of the reduction rules for our new constructs; the others ones are standard, and so are the congruence rules that reduce subexpressions, e.g. $\rho \vdash C, \mathtt{tell}(e) \to C', \mathtt{tell}(e')$ if $\rho \vdash C, e \to C', e'$.

We briefly comment below on the rules displayed. 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

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

(DLET1)
$$\frac{\rho[(G.e_1, \rho(\tilde{x}))/\tilde{x}] \vdash C, e_2 \rightarrow C', e_2'}{\rho \vdash C, dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2 \rightarrow C', dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2'}$$

(DLET2)
$$\frac{}{\rho \vdash C, dlet\, \tilde{x} = e_1\, when\, G\, in\, v \rightarrow C, v}$$

(PAR)
$$\frac{\rho(\tilde{x}) = Va \qquad dsp(C, Va) = (e, \theta)}{\rho \vdash C, \tilde{x} \rightarrow C, e\theta}$$

(TELL2)
$$\frac{dsp(C \cup \{F\}, phi.()) = ((), \emptyset)}{\rho \vdash C, tell(F)^l \rightarrow C \cup \{F\}, ()}$$

(VAAPP3)
$$\frac{dsp(C, Va) = (e, \{\overrightarrow{c}/\overrightarrow{y}\})}{\rho \vdash C, \#((x)\{Va\}, v) \rightarrow C, e\{v/x, \overrightarrow{c}/\overrightarrow{y}\}}$$

(RETRACT2)
$$\frac{dsp(C \smallsetminus \{F\}, phi.()) = ((), \emptyset)}{\rho \vdash C, retract(F)^l \rightarrow C \smallsetminus \{F\}, ()}$$

Fig. 2. The reduction rules for the constructs of $\mathrm{ML}_{\mathrm{CoDa}}$ concerning adaptation

It inspects a variation from left to right to find the first goal G satisfied by C, under a substitution θ. If this search succeeds, the dispatching returns the corresponding expression e and θ. Then, $\tilde{x}$ reduces to $e\theta$, i.e. to e the variables of which 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.

Consider the simple conditional expression $\texttt{if}\, \tilde{x} = F_2\, \texttt{then}\, 42\, \texttt{else}\, 51$, in an environment ρ that binds the parameter $\tilde{x}$ to $e' = \texttt{G}_1.\texttt{F}_5, \texttt{G}_2.\texttt{F}_2$ and in a context C that satisfies the goal G_2, but not G_1:

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

where we first retrieve the binding for $\tilde{\texttt{x}}$ (recall it is e'), with $dsp(C, e') = (\texttt{F}_2, \theta)$, 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. Consider the behavioural variation $(x)\{G_1.c_1, G_2.x\}$ and apply it to the constant c in a context C that satisfies the goal G_2, but not G_1. Since $dsp(C, (x)\{G_1.c_1, G_2.x\}) = (x, \theta)$ for some substitution θ, we get

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

The rule for $tell(e)^l/retract(e)^l$ evaluates the expression e until it reduces to a fact F, which is a value of $\mathrm{ML}_{\mathrm{CoDa}}$. The new context C', obtained from C by adding/removing F, is checked against the security policy Φ. Since Φ is a Datalog goal, we can easily reuse our dispatching machinery, implementing the check as a call to the function dsp where the first argument is C' and the second one is the trivial variation $\texttt{phi}.()$. 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* construct. Let Φ be the policy of Sect. 2, C be $\{F_3, F_4, F_5\}$, and apply $\texttt{f} = \lambda\texttt{x}.\,\texttt{if}\,\texttt{e}_1\,\texttt{then}\,\texttt{F}_5\,\texttt{else}\,\texttt{F}_4$ to unit. If

e_1 evaluates to `false` (without changing the context), the evaluation gets stuck because $dsp(C \smallsetminus \{F_4\}, \texttt{phi.}())$ fails. Since Φ requires the fact F_4 to always hold, every attempt to remove it from the context violates indeed Φ. If, instead, e_1 reduces to `true`, there is no policy violation and the evaluation reduces to unit.

$$\rho \vdash C, \texttt{retract(f ())}^l \to^* C, \texttt{retract(F}_4\texttt{)}^l \not\to$$
$$\rho \vdash C, \texttt{retract(f ())}^l \to^* C, \texttt{retract(F}_5\texttt{)}^l \to C \smallsetminus \{\texttt{F}_5\}, ()$$

4 Type and Effect System

We now associate an $\mathrm{ML}_{\mathrm{CoDa}}$ expression with a type, an abstraction called the *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 runtime. Then, the labelling environment drives code instrumentation with security checks. First, we briefly present History Expressions and labelling environments, and then the rules of our type and effect system.

History Expressions A history expression is a term of a simple process algebra that soundly abstracts program behaviour [4]. Here, they approximate the sequence of actions that an application may perform over the context at runtime, i.e. asserting/retracting facts and asking if a goal holds. We assume that a history expression is uniquely labelled on a given set of Lab_H. Labels link static actions in histories to the corresponding dynamic actions inside the code. The syntax of History Expressions follows:

$$H ::= \ni \mid \varepsilon^l \mid h^l \mid (\mu h.H)^l \mid tell\,F^l \mid retract\,F^l \mid (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \Delta$$
$$\Delta ::= (ask\,G.H \otimes \Delta)^l \mid fail^l$$

The empty history expression abstracts programs which do not interact with the context. For technical reasons, we syntactically distinguish when the empty history expression comes from the syntax (ε^l) and when it is instead obtained by reduction in the semantics ($\ni$). With $\mu h.H$ we represent possibly recursive functions, where h is the recursion variable; the "atomic" history expressions $tell\,F$ and $retract\,F$ 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; Δ mimics our dispatching mechanism, where Δ is an *abstract variation*, defined as a list of history expressions, each element H_i of which is guarded by an $ask\,G_i$.

The history expression of the behavioural variation `url` in `getExhibitData` of Sect. 2, is $H_{url} = ask\,G_1.H_1 \otimes ask\,G_2.H_2 \otimes fail$, where $G_1 = \leftarrow \texttt{direct\_comm}()$ and $G_2 = \leftarrow \texttt{use\_qrcode (decoder)}, \texttt{camera(cam)}$, and H_i is the effect of the expression guarded by G_i, for $i = 1, 2$. Intuitively, H_{url} means that at least one goal between G_1 and G_2 must be satisfied by the context to successfully apply the behavioural variation `url`. Given a context C, the behaviour of a history expression H is formalised by the transition system inductively defined in Fig. 3. Transitions $C, H \to C', H'$ mean that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones of [4]: below we only comment on those dealing with the context. An action $tell\,F$

$$\frac{}{C,\,(\ni\cdot H)^l \to C, H} \qquad \frac{}{C, tell\,F^l \to C\cup\{F\},\,\ni} \qquad \frac{C, H_1 \to C', H_1'}{C, (H_1+H_2)^l \to C', H_1'}$$

$$\frac{}{C,\,\varepsilon^l \to C,\,\ni} \qquad \frac{}{C, retract\,F^l \to C\backslash\{F\},\,\ni} \qquad \frac{C, H_2 \to C', H_2'}{C, (H_1+H_2)^l \to C', H_2'}$$

$$\frac{C, H_1 \to C', H_1'}{C,\,(H_1\cdot H_2)^l \to C',\,(H_1'\cdot H_2)^l} \qquad \frac{C \vDash G}{C,\,(ask\,G.H\otimes\Delta)^l \to C, H}$$

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

Fig. 3. Semantics of History Expressions

reduces to $\ni$ and yields a context C', where the fact F has just been added; similarly for $retract\,F$. Differently from what we do in the semantic rules, here we do not consider the possibility of a policy violation: a history expression approximates how the application would behave in the absence of any kind of check. The rules for Δ scan the abstract variation and look for the first goal G satisfied in the current context; if this search succeeds, the whole history expression reduces to the history expression H guarded by G; otherwise the search continues on the rest of Δ. If no satisfiable goal exists, the stuck configuration *fail* is reached, meaning that the dispatching mechanism fails.

We assume we are given the function $h : Lab_H \to \mathbb{H}$ that recovers a construct in a given history expression $h \in \mathbb{H}$ from a label l. Below, we specify the link between a $tell/retract$ in a history expression and its corresponding operation in the code, labelled on Lab_C (see Sect 3). Consider, e.g. the history expression H_a of Sect. 2, and the label correspondence given there: $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$.

Definition 1 (Labelling environment). *A* labelling environment *is a (partial) function* $\Lambda : Lab_H \to Lab_C$*, defined only if* $h(l) \in \{tell(F), retract(F)\}$.

Typing Rules We assume that each Datalog predicate has a fixed arity and a type (see [20]). 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 systems 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 does not occur in Γ).
- a further environment $K ::= \emptyset \mid K, (\tilde{x}, \tau, \Delta)$, that maps a parameter $\tilde{x}$ to a pair consisting of a type and an abstract variation Δ, used to solve the binding for $\tilde{x}$ at runtime; $K, (\tilde{x}, \tau, \Delta)$ denotes an environment with a binding for the parameter $\tilde{x}$ (not in K).

Our typing judgements $\Gamma; K \vdash e : \tau \triangleright H; \Lambda$, express that in the environments Γ and K the expression e has type τ, effect H and yields a labelling environment Λ. We have basic types $\tau_c \in \{int, bool, unit, \ldots\}$, functional types, behavioural variations types, and facts:

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

$$\text{(TFACT)}\quad \frac{}{\Gamma; K \vdash F \,:\, fact_{\{F\}} \triangleright \varepsilon; \bot}$$

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

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

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

$$\text{(TDLET)}\quad \frac{\begin{array}{c} \Gamma, \overrightarrow{y} \,:\, \overrightarrow{\tilde{\tau}}; K \vdash e_1 \,:\, \tau_1 \triangleright H_1; \Lambda_1 \\ \Gamma; K, (\tilde{x}, \tau_1, \Delta') \vdash e_2 \,:\, \tau \triangleright H; \Lambda_2 \end{array}}{\Gamma; K \vdash dlet\, \tilde{x} = e_1\, when\, G\, in\, e_2 \,:\, \tau \triangleright H; \Lambda_1 \uplus \Lambda_2} \qquad \begin{array}{l} \text{where } \gamma(G) = \overrightarrow{y} \,:\, \overrightarrow{\tilde{\tau}} \\ \text{if } K(\tilde{x}) = (\tau_1, \Delta) \text{ then } \Delta' = G.H_1 \otimes \Delta \\ \text{else (if } \tilde{x} \notin K \text{ then } \Delta' = G.H_1 \otimes fail) \end{array}$$

Fig. 4. Typing rules for the new constructs implementing adaptation

Some types are annotated for analysis reasons. In $fact_\phi$, the set ϕ contains the facts that an expression can be reduced to at runtime (see the semantics rules (TELL2) and (RETRACT2)). Here, K stores the types and the abstract variations of the parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e. the sequence of actions which may be performed over the context during the function evaluation. Similarly, in $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv, while Δ is an abstract variation, that represents the information used at runtime by the dispatching mechanism to apply bv.

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

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

Most of the rules of our type and effect system are inherited from ML, and those for the new constructs are in Fig. 4. A few comments are in order.

The rule (TFACT) gives a fact F type *fact* annotated with $\{F\}$ and the empty effect. The rule (TTELL)/(TRETRACT) asserts that the expression $tell(e)/retract(e)$ has type *unit*, provided that the type of e is $fact_\phi$. The overall effect is obtained by combining the effect of e with the nondeterministic summation of $tell\,F/retract\,F$, where F is any of the facts in the type of e. In rule (TVARIATION) we determine the type for each subexpression e_i under K', and the environment Γ, extended by the type of x and of the variables $\overrightarrow{y_i}$ occurring in the goal G_i (recall that the Datalog typing function γ returns a list of pairs (z, type-of-z) for all variables z of G_i). Note that all subexpressions e_i have the same type τ_2. We also require that the abstract variation Δ results from concatenating $ask\, G_i$ with the effect computed for e_i. The type of the behavioural variation is annotated by K' and Δ. Consider the behavioural variation $bv_1 = (x)\{G_1.e_1, G_2.e_2\}$. Assume that the two cases of this behavioural variation have type τ and effects H_1 and H_2, respectively, under the environment $\Gamma, x : int$ (goals have no variables) and the guessed environment K'. Hence, the type of bv_1 will be $int \xrightarrow{K'|\Delta} \tau$ with $\Delta = ask\, G_1.H_1 \otimes ask\, G_2.H_2 \otimes fail$ and the effect will be empty. The rule (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. The labelling environment generated by the rules (TFACT) is $\perp$, because there is no $tell$ or $retract$. Instead both (TTELL) and (TRETRACT) update the current environment Λ by associating all the labels of the facts which e can evaluate to, with the label l of the $tell(e)$ ($retract(e)$, resp.) being typed. The rule (TLET) produces an environment Λ that contains all the correspondences of Λ_1 and Λ_2 coming from e_1 and e_2; note that unicity of the labelling is guaranteed by the condition $dom(\Lambda_1) \cap dom(\Lambda_2) = \emptyset$.

The correspondence between the labels in the expression e_a and those of its history expression H_a of Sect. 2 are $\{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 3, 5 \mapsto 4, 8 \mapsto 5\}$, and the other labels are mapped to $\perp$. Note that a labelling environment need not be injective.

Soundness Our type and effect system is sound with respect to the operational semantics of $\mathrm{ML}_{\mathrm{CoDa}}$. First, we introduce the typing dynamic environment and an ordering on history expressions. Intuitively, the history expression H_1 could be obtained from H_2 by evaluation.

Definition 2 (Typing dynamic environment). *Given the type environments Γ and K, we say that the dynamic environment ρ has type K under Γ (in symbols $\Gamma \vdash \rho : K$) iff $dom(\rho) \subseteq dom(K)$ and $\forall \tilde{x} \in dom(\rho)\,.\ \rho(x) = G_1.e_1, \dots, G_n.e_n$ $K(\tilde{x}) = (\tau, \Delta)$ and $\forall i \in \{1, \dots, n\}\,.\ \gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K_{\tilde{x}} \vdash e_i : \tau' \triangleright H_i$ and $\tau' \leq \tau$ and $\bigotimes_{i \in \{1,\dots,n\}} G_i.H_i \sqsubseteq \Delta$.*

Definition 3. *Given H_1, H_2 then $H_1 \preccurlyeq H_2$ iff one of the following cases holds*

(a) $H_1 \sqsubseteq H_2$; *(b) $H_2 = H_3 \cdot H_1$ for some H_3;*
(c) $H_2 = \bigotimes_{i \in \{1,\dots,n\}} ask\, G_i.H_i \otimes fail \wedge H_1 = H_i$, $i \in [1..n]$.

Theorem 1 (Preservation). *Let e_s be a closed expression; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s and such that $\Gamma \vdash \rho : K$. If $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$ and $\rho \vdash C, e_s \to C', e'_s$ then*
$\Gamma; K \vdash e'_s : \tau \triangleright H'_s; \Lambda'_s$ and $\exists \overline{H}$, s.t. $\overline{H} \cdot H'_s \preccurlyeq H_s$ and $C, \overline{H} \cdot H'_s \to^+ C', H'_s$ and $\Lambda'_s \sqsubseteq \Lambda_s$.

The Progress Theorem assumes that the effect H is *viable*, i.e. it does not reach *fail*, meaning that the dispatching mechanism succeeds at runtime. The control flow analysis of Sect. 5 guarantees viability (below $\rho \vdash C, e \not\to$ means no transition from C, e). 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.

Theorem 2 (Progress). *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s; \Lambda_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho : K$. If $\rho \vdash C, e_s \nrightarrow$ and H is viable for C (i.e. $C, H_s \nrightarrow^+ C', fail$) and there is no policy violation then e_s is a value.*

Corollary 1 (Over-approximation). *Let e be a closed expression.*
If $\Gamma; K \vdash e : \tau \triangleright H; \Lambda_s \ \wedge\ \rho \vdash C, e \rightarrow^* C', e'$, for some ρ such that $\Gamma \vdash \rho : K$, then there exists a computation $C, H \rightarrow^* C', H'$, for some H'.

Note that the type of e' is the same of e, because of Theorem 1, and the obtained label environment is included in Λ_s.

5 Loading-time Analysis

Our execution model for $\mathrm{ML_{CoDa}}$ extends the one in [13]: the compiler produces a quadruple $(C_p, e_p, H_p, \Lambda_p)$ given by the application context, the object code, the history expression over-approximating the behaviour of e_p; and the labelling environment associating labels of H_p with those in the code. Given the quadruple, at loading time, the virtual machine performs the following two phases:

– *linking*: to resolve system variables and constructs the initial context C (combining C_p and the system context); and
– *verification*: to build from H_p a graph $\mathcal{G}$ that describes the possible evolutions of C.

Technically, we compute $\mathcal{G}$ through a static analysis, specified in terms of Flow Logic [21]. To support the formal development, we assume below that all the bound variables occurring in a history expression are distinct. So we can define a function $\mathbb{K}$ mapping a variable h^l to the history expression $(\mu h.H_1^{l_1})^{l_2}$ that introduces it.

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

– the *pre-set* $\Sigma_\circ(l)$ contains the contexts possibly arising *before* evaluating H^l;
– the *post-set* $\Sigma_\bullet(l)$ contains the contexts possibly resulting *after* evaluating H^l.

The analysis is specified by a set of clauses upon judgements $(\Sigma_\circ, \Sigma_\bullet) \models H^l$, where $\models \subseteq \mathcal{AE} \times \mathbb{H}$ and $\mathcal{AE} = (Lab_H \rightarrow \wp(Context \cup \{\bullet\}))^2$ is the domain of the results of the analysis and $\mathbb{H}$ the set of history expressions. The judgement $(\Sigma_\circ, \Sigma_\bullet) \models H^l$ says that $\Sigma_\circ$ and $\Sigma_\bullet$ form an acceptable analysis estimate for the history expression H^l.

We will use the notion of acceptability to check whether the history expression H_p, hence the expression e it is an abstraction of, will never fail in a given initial context C.

In Fig. 5, we give the set of inference rules that validate the correctness of a given estimate $\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet)$. Intuitively, the checks in the clauses mimic the semantic evolution of the history expression in the given context, by modelling the semantic preconditions and the consequences of the possible reductions.

In the rule (ATELL), the analysis checks whether the context C is in the pre-set, and $C \cup \{F\}$ is in the post-set; similarly for(ARETRACT), where $C \backslash \{F\}$ should be in

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

$$\text{(ASEQ1)}\quad \frac{\begin{array}{ll}(\Sigma_\circ,\Sigma_\bullet)\models H_1^{l_1}\;l & \Sigma_\circ(l)\subseteq\Sigma_\circ(l_1)\\ (\Sigma_\circ,\Sigma_\bullet)\models H_2^{l_2} & \Sigma_\bullet(l_1)\subseteq\Sigma_\circ(l_2)\\ & \Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models (H_1^{l_1}\cdot H_2^{l_2})^l} \qquad \text{(ASEQ2)}\quad \frac{(\Sigma_\circ,\Sigma_\bullet)\models H_2^{l_2} \quad \begin{array}{l}\Sigma_\circ(l)\subseteq\Sigma_\circ(l_2)\\ \Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models (\ni\cdot H_2^{l_2})^l}$$

$$\text{(AEPS)}\quad \frac{\Sigma_\circ(l)\subseteq\Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet)\models \varepsilon^l} \qquad \text{(ASUM)}\quad \frac{\begin{array}{lll}(\Sigma_\circ,\Sigma_\bullet)\models H_1^{l_1} & \Sigma_\circ(l)\subseteq\Sigma_\circ(l_1) & \Sigma_\bullet(l_1)\subseteq\Sigma_\bullet(l)\\ (\Sigma_\circ,\Sigma_\bullet)\models H_2^{l_2} & \Sigma_\circ(l)\subseteq\Sigma_\circ(l_2) & \Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models (H_1^{l_1}+H_2^{l_2})^l}$$

$$\text{(AASK1)}\quad \frac{\begin{array}{ll}\forall C\in\Sigma_\circ(l) & (C\models G \implies (\Sigma_\circ,\Sigma_\bullet)\models H^{l_1} \quad \Sigma_\circ(l)\subseteq\Sigma_\circ(l_1) \quad \Sigma_\bullet(l_1)\subseteq\Sigma_\bullet(l))\\ & (C\not\models G \implies (\Sigma_\circ,\Sigma_\bullet)\models \Delta^{l_2} \quad \Sigma_\circ(l)\subseteq\Sigma_\circ(l_2) \quad \Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l))\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models (\mathit{ask}G.H^{l_1}\otimes\Delta^{l_2})^l} \qquad \text{(AASK2)}\quad \frac{\bullet\in\Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet)\models \mathit{fail}^l}$$

$$\text{(AREC)}\quad \frac{(\Sigma_\circ,\Sigma_\bullet)\models H^{l_1} \quad \begin{array}{l}\Sigma_\circ(l)\subseteq\Sigma_\circ(l_1)\\ \Sigma_\bullet(l_1)\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models (\mu h.H^{l_1})^l} \qquad \text{(AVAR)}\quad \frac{\mathbb{K}(h)=(\mu h.H^{l_1})^{l'} \quad \begin{array}{l}(\Sigma_\circ(l)\subseteq\Sigma_\circ(l')\\ \Sigma_\bullet(l')\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\models h^l}$$

Fig. 5. Specification of the analysis for History Expressions

the post-set. The rule (ANIL) says that every pair of functions is an acceptable estimate for the "semantic" empty history expression $\ni$. The estimate $\mathcal{E}$ is acceptable for the "syntactic" ε^l if the pre-set is included in the post-set (rule (AEPS)). The rules (ASEQ1) and (ASEQ2) handle the sequential composition of history expressions. The first rule states that $(\Sigma_\circ,\Sigma_\bullet)$ is acceptable for $H=(H_1^{l_1}\cdot H_2^{l_2})^l$ if it is valid for both H_1 and H_2. Moreover, the pre-set of H_1 must include the pre-set of H and the pre-set of H_2 includes the post-set of H_1; finally, the post-set of H includes that of H_2. The second rule states that $\mathcal{E}$ is acceptable for $H=(\ni\cdot H_1^{l_2})^l$ if it is acceptable for H_1 and the pre-set of H_1 includes that of H, while the post-set of H includes that of H_1. The rules (AASK1) and (AASK2) handle the abstract dispatching mechanism. The first states that $\mathcal{E}$ is acceptable for $H=(\mathit{ask}G.H_1^{l_1}\otimes\Delta^{l_2})^l$, provided that, for all C in the pre-set of H, if the goal G succeeds in C then the pre-set of H_1 includes that of H and the post-set of H includes that of H_1. Otherwise, the pre-set of Δ^{l_2} must include the pre-set of H and the post-set of Δ^{l_2} is included in that of H. The second requires $\bullet$ to be in the post-set of fail^l. 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-set of H is included in the pre-sets of H_1 and H_2; and the post-set of H includes those of H_1 and H_2. 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-set of H_1 includes that of H; and the post-set of H includes that of H_1.

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

Semantic properties We now formalise the notion of valid estimate for a history expression; we prove that there always exists a minimal valid analysis estimate; and that a valid estimate is correct w.r.t. the operational semantics of history expressions.

Definition 4 (Valid analysis estimate). *Given $H_p^{l_p}$ and an initial context C, we say that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is a* valid analysis estimate *for H_p and C iff $C \in \Sigma_\circ(l_p)$ and $(\Sigma_\circ, \Sigma_\bullet) \models H_p^{l_p}$.*

Theorem 3 (Existence of estimates). *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \models H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

Theorem 4 (Subject Reduction). *Let H^l be a closed history expression s.t. $(\Sigma_\circ, \Sigma_\bullet) \models H^l$. If $\forall C \in \Sigma_\circ(l)$, $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \models H'^{l'}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$, and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.*

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

Definition 5 (Viability). *Let H_p be a history expression and C be an initial context. We say that H_p is* viable *for C if there exists the minimal valid analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ such that $\forall l \in dom(\Sigma_\bullet) \backslash lfail(H_P)\ \bullet \notin \Sigma_\bullet(l)$.*

To illustrate how viability is checked, consider the following history expressions:

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

and the initial context $C = \{F_2, F_5, F_8\}$, only consisting of facts.

The left part of Fig. 6 shows the values of $\Sigma_\circ^1(l)$ and $\Sigma_\bullet^1(l)$ for H_p. Notice that the pre-set of $tell\,F_1^1$ includes $\{F_2, F_5, F_8\}$, and the post-set also includes $\{F_1\}$. Also, the pre-set of $retract\,F_8^5$ includes $\{F_2, F_5, F_8\}$, while the post-set includes $\{F_2, F_5\}$. The column describing $\Sigma_\bullet$ contains $\bullet$ only for $l = 7$, the label of *fail*, so H_p is viable for C. However, the history expression H_p' fails to pass the verification phase, when put in the initial context C. Since the goal F_3 does not hold in C, H_p' is not viable. This is reflected by the occurrences of $\bullet$ in $\Sigma_\bullet^2(4)$ and $\Sigma_\bullet^2(7)$, as shown in the right part of Fig. 6.

Now, we exploit the result of the above analysis to build up the evolution graph $\mathcal{G}$. It describes how the initial context C will evolve at runtime, paving the way to security enforcement.

Definition 6 (Evolution Graph). *Let H_p be a history expression, C be a context, and $(\Sigma_\circ, \Sigma_\bullet)$ be a valid analysis estimate. The evolution graph of C is $\mathcal{G} = (N, E, L)$, where*

$$\begin{array}{lcl}
N & = & \bigcup_{l \in Lab_H^*} (\Sigma_\circ(l) \cup \Sigma_\bullet(l)) \\
E & = & \{(C_1, C_2) \mid \exists F \in Fact^*,\ l \in Lab_H^*\ s.t.\ C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge \\
 & & (h(l) \in \{tell(F), retract(F)\} \vee (C_2 = \bullet))\} \\
L & : & E \to \mathcal{P}(Labels) \\
\forall t & = & (C_1, C_2) \in E,\ l \in L(t)\ iff\ C_1 \in \Sigma_\circ(l) \wedge C_2 \in \Sigma_\bullet(l) \wedge h(l) \neq fail
\end{array}$$

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

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

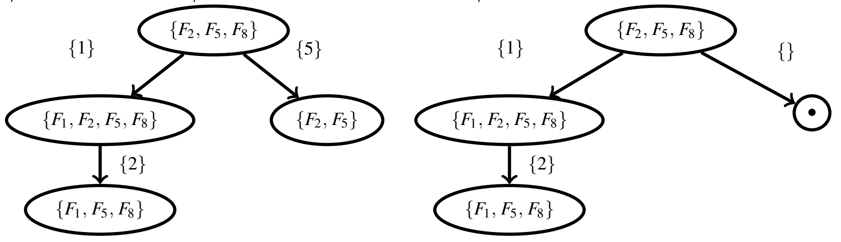

Fig. 6. The analysis results (top) and the evolution graphs $\mathcal{G}_p$ (bottom left) and $\mathcal{G}'_p$ (bottom right) for the initial context $C = \{F_2, F_5, F_8\}$, and for the history expressions $H_p = ((tell\,F_1^1 \cdot retract\,F_2^2)^3 + (ask\,F_5.retract\,F_8^5 \otimes ask\,F_3.retract\,F_4^6 \otimes fail^7)^4)^8$ and $H'_p = ((tell\,F_1^1 \cdot retract\,F_2^2)^3 + (ask\,F_3.retract\,F_4^5 \otimes fail^6)^4)^7$, respectively.

Intuitively, the nodes of $\mathcal{G}$ are sets of contexts, and an edge between two nodes C_1 and C_2 records that C_2 is obtained from C_1, through a $tell/retract$. Using the labels of arcs we can locate the abstract $tell/retract$ that may lead to a context violating a given policy Φ. By putting guards on the corresponding risky actions in the code (via Λ), we can enforce Φ. In the following, let $Fact^*$ and Lab^*_H be the set of facts and the set of labels occurring in H_p, i.e. the history expression under verification.

Consider again the history expressions H_p and H'_p and their evolution graphs $\mathcal{G}_p$ and $\mathcal{G}'_p$ (Fig. 6, bottom). In $\mathcal{G}_p$, from the initial context C there is an arc labelled $\{1\}$ to $C \cup \{F_1\}$, because of $tell\,F_1^1$, and there is an arc labelled $\{5\}$ to the $C \setminus F_8$, because of $retract\,F_8^5$. It is easy to see that H_p is viable for C, because the node $\bullet$ is not reachable from the initial context C in $\mathcal{G}_p$, However H'_p is not, because $\bullet$ is reachable in $\mathcal{G}'_p$.

6 Code instrumentation

Once we detected the potentially risky operations through the evolution graph $\mathcal{G}$, we can instrument the code of an application e and only switch on our *runtime monitor* to guard them. First, since a node n of $\mathcal{G}$ represents a context reachable while executing e, we *statically* verify whether n satisfies Φ. If this is not the case, we consider all the edges with target n and the set R of their labels. The labelling environment Λ, computed while type checking e, determines those actions in the code that require monitoring during the execution, indexed by the set $Risky = \Lambda(R)$.

In fact we will guard all the $tell/retract$ actions in the code, but our runtime monitor will only be invoked on the risky ones. To do that, the compiler labels the source code as said in Sect. 2 and generates specific calls to *monitoring* procedures. We offer a lightweight form of code instrumentation that does not operate on the object code, differently from standard instrumentation. In more detail, we define a procedure, called `check_violation(l)`, for verifying if the policy Φ is satisfied. It takes a label `l` as parameter and returns the type *unit*. At loading time, we assign a global mask `risky[l]` for each label `l` in the source code, by using the information in the set *Risky*.
The procedure code in a pseudo ML$_{\text{CoDa}}$ and the definition of `risky[l]` are as follows:

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

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

If `risky[l]` is `false`, then the procedure returns to the caller and the execution goes on normally. Otherwise, it calls for a check on Φ, by triggering with the call `ask phi.()` the dispatching mechanism: if the call fails then a policy violation is about to occur. In this case the computation is aborted or a recovery mechanism is possibly invoked.

Our compilation schema needs to replace every $tell(e)^l$ (similarly for $retract(e)^l$) in the source code with the following, where `z` is fresh.:

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

An easy optimisation is possible when *Risky* is empty, i.e. when the analysis ensures that all the $tell/retract$ actions are safe and so no execution paths lead to a policy violation. To do this, we introduce the flag `always_ok`, whose value will be computed at linking time: if it is true, no check is needed. The previous compilation schema is simply refined by testing `always_ok` before calling `check_violation`.

7 Conclusions

We have addressed security issues in an adaptive framework, by extending ML$_{\text{CoDa}}$, a functional language introduced in [13] for adaptive programming. Our main contributions can be summarised as follows.

– We have expressed and enforced *context-dependent* security policies in Datalog, originally used by ML$_{\text{CoDa}}$ to deal with contexts.
– We have extended the ML$_{\text{CoDa}}$ type and effect system for computing a type and a labelled abstract representation of the overall behaviour of an application. Actually, an effect over-approximates the sequences of the possible dynamic actions over the context, and labels link the security-critical operations of the abstraction with those in the code of the application.

– We have enhanced the static analysis of [13] to identify the operations that may affect contexts and violate the required *policy*, besides verifying that the application can adapt to all the possible contexts arising at runtime.
– Based on the results of the static analysis, we have defined a way to instrument the code of an application *e*, so as to introduce an *adaptive runtime monitor* that stops *e* when about to violate the policy to be enforced, and is switched on and off at need.

We plan to investigate richer forms of policies, in particular those having an additional dynamic scope, and those which are history dependent [4], and study their impact on adaptivity. A long term goal is extending these policies with quantitative information, e.g. statistical information about the usage of contexts, reliability of resources therein, etc. Finally, we are thinking of providing a kind of recovery mechanism for behavioural variations, to allow the user to undo some actions considered risky or sensible, and force the dispatching mechanism to make different, alternative choices.

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. Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H.: ContextJ: Context-oriented programming with Java. Computer Software 28(1) (2011)
4. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6) (2009)
5. 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)
6. 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)
7. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. 240(1), 177–213 (2000)
8. 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)
9. Costanza, P.: Language constructs for context-oriented programming. In: Proc. of the Dynamic Languages Symposium. pp. 1–10. ACM Press (2005)
10. 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)
11. 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)
12. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems 5(1), 1–35 (1997)
13. Galletta, L.: Adaptivity: linguistic mechanisms and static analysis techniques. Ph.D. thesis, University of Pisa (2014), `http://www.di.unipi.it/~galletta/phdThesis.pdf`
14. 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)
15. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology, March-April 2008 7(3), 125–151 (2008)

16. 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)
17. 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)
18. 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)
19. 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)
20. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for prolog. Artificial Intelligence 23(3), 295 – 307 (1984)
21. 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)
22. 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)
23. Pasquale, L., Ghezzi, C., Menghi, C., Tsigkanos, C., Nuseibeh, B.: Topology Aware Adaptive Security, to appear in SEAMS 2014
24. Pfleeger, C., Pfleeger, S.: Security in computing. Prentice Hall (2003)
25. 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)
26. 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)
27. 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)