Last Mile's Resources

Chiara Bodei(✉), Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta

Dipartimento di Informatica, Università di Pisa, Pisa, Italy
{chiara,degano,giangi,galletta}@di.unipi.it

Abstract. We extend an existing two-phase static analysis for an adaptive programming language to also deal with dynamic resources. The focus of our analysis is on predicting how these are used, in spite of the different, ever changing operating environments to which applications automatically adapt their behaviour. Our approach is based on a type and effect system at compile time, followed by a control flow analysis carried on at loading time. Remarkably, the second analysis cannot be anticipated, because information about availability, implementation and other aspects of resources are unknown until the application is injected in the current environment.

1 Introduction

Today's software systems are expected to operate *every time*, *everywhere* within a *highly dynamic and open operational environment*. Also *software is eating the world* by pervading the objects of our everyday life, such as webTV, coffeemakers, wearable devices, cars, smartphones, ebook readers, and Smart Cities, on a broader scale. The operational environment of software systems, often referred to as the *context*, has indeed turned to be a *virtual computing platform* that provides access to groups of heterogeneous *smart* resources. The main distinguishing characteristic of these resources is that in principle they are always connected to the Internet, possibly linking to it through different access points, so to coordinate and interact each other. For instance, your smart alarm clock can activate your coffeemaker to prepare you a cup of coffee. In general, the Internet of Things scenario is the most significant example of this computing framework. In this vision, the physical resources (e.g. the coffeemaker) are difficult to tell apart from the virtual ones (e.g. your wireless network), in that they are potentially unlimited and "virtualised" in order to appear fully dedicated to their users. In addition, they can choose on their own *where, when* and to *whom* they are visible and in *which* portions of their context. A further important feature is that smart resources can collect and exchange information of various kind. According to their knowledge, smart resources can perform actions that also modify the environment.

A key challenge is designing software systems that run without compromising their intended behaviour or their non-functional requirements, e.g. quality of

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

C.W. Probst et al. (Eds.): Nielsons' Festschrift, LNCS 9560, pp. 33–53, 2016.
DOI: 10.1007/978-3-319-27810-0_2

service, when injected in highly dynamic and open operational environments. Programming these systems thus requires new programming language features and effective mechanisms to sense the modifications of the actual context and to properly *adapt* to changes. We refer to [2] for a comprehensive discussion on the software engineering challenges of the *open world assumption* and *adaptation*.

Several approaches have been considered for adapting applications according to the resources currently available in their running context, and to the form these resources assume. In this paper, we address this issue from a language-based perspective, by relying on $\mathrm{ML}_{\mathrm{CoDa}}$ [5,8,9], a core ML with Context-Oriented Programming features [13,17]. Its main novelty is to be a two-component language: a declarative part for handling the context and a functional one for computing. The context in $\mathrm{ML}_{\mathrm{CoDa}}$ is a knowledge base implemented in Datalog, and queries to the context return information about resources, as well as handles to access them.

The $\mathrm{ML}_{\mathrm{CoDa}}$ context has been designed to hide the complexity of the operational environment and to provide an abstraction from low level details such as protocol handling, data marshalling, networking technologies, and so on. Consequently, it masks the heterogeneity of the virtual computing infrastructure to facilitate the design and the development of applications.

As usual, applications are assumed to know in advance the kind of smart resources that are possibly hosted in the environment. However, the resources that are actually present in the context and their form can only be discovered when the application enters the context and is about to run. Technically, each resource is supposed to come equipped with a resource manager and with a public API. An application can manipulate a resource through a *handle* provided by its manager, that also governs the life-cycle of the resource, its availability, etc. The handle enables the application to operate over a resource through the mechanisms declared by the API. Actually, this contains information about the kind of the resource, such as the available operations, their signatures etc. In particular, by querying the context, an $\mathrm{ML}_{\mathrm{CoDa}}$ application can operate on a resource both *explicitly* by retrieving its handle, and *implicitly* by inspecting its current status through system predicates. For example, a home automation system controlling the house can query the alarm clock to retrieve the level of battery. In the $\mathrm{ML}_{\mathrm{CoDa}}$ programming model, the running context consists of two parts: the *system* and the *application* context. The first one is provided by the $\mathrm{ML}_{\mathrm{CoDa}}$ virtual machine through its API, while the other one stores specific knowledge of the application, filled in by the programmer. In the execution model the actual state of a resource, as well as its usage constraints, is completely known only at runtime. A relevant goal is therefore to ensure that an application that enters into a context finds all the needed resources and uses them correctly. This assurance can offer the basis for providing highly "reliable" service management for virtual computing platforms such as the Internet of Things.

In this paper we suitably extend the two-step static analysis of [9] to take care of resources. We call our proposal *last mile* analysis, right because full knowledge on the context is only available at runtime.

Technically, the ML_{CoDa} compiler produces a triple (C, e, H) consisting of the application context, the object code and an effect over-approximating the behaviour of the application. The third component H describes resource usage, independently of the possible context running the application, so it contains parameters to be instantiated when available. Using the above triple, the ML_{CoDa} virtual machine performs a linking and a verification phase at loading time: the last mile. During the linking phase, the initial context is constructed, by merging the application and the system contexts. Then the verification phase checks whether the application adapts to all evolutions of the initial context that may occur at runtime (a *functional* property), and whether it respects the constraints on the usage of the resources (a *non-functional* property). Only programs that pass this verification phase will be run.

To efficiently perform the last mile analysis, we build a graph $\mathcal{G}$ describing the possible evolutions of the initial context. Technically, we compute $\mathcal{G}$ from H, through a static analysis specified in terms of Flow Logic [14,16]. The evolution graph facilitates checking functional and non-functional properties, reducing them to reachability. The non-functional properties are similar to those expressible in CTL* [1], in that they predicate over nodes, i.e. contexts, and paths of $\mathcal{G}$.

This paper is organised as follows. The next section intuitively introduces ML_{CoDa} and our approach, with the help of some illustrative examples. The syntax and the operational semantics of our extension of ML_{CoDa} are formally given in Sect. 3. The next two sections describe our two-phase static analysis: Sect. 4 presents the type and effect system, while Sect. 5 presents the loading time analysis. Section 6, summarises our results and discusses some future work.

2 An Example

Consider a mobile application used for accessing to some databases of a company. The vendors, among which Jane and Bob, can access the databases from both inside and outside the office. The access control policies are part of the context of the application, so they are stored as Datalog facts and predicates. For example, 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,db) ← delegate(z,x,db), is_admin(z)
```

The context typically includes other information ranging on a wide collection, e.g. users, administrators, location of users and company offices, information about the company ICT services, etc. Also, the context contains information about the application state, e.g. the application is connected to the database through the company intranet or through an external proxy.

The following ML_{CoDa} code implements a simple application which accesses a database and performs a query to retrieve data about customers. The execution

depends on the location and on the capabilities 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.

```
1  fun main () =
2    let records = (table){
3      ← office(),current_usr(name),has_auth(name,handle).
4          let c = open_db(handle) in
5            query(c, select * from table)
6      ← ¬office(),current_usr(name),has_auth(name,handle),
7         proxy(ip),crypto_key(k).
8          let chan = connect(ip) in
9          let c = get_db(chan) in
10         let data = crypto_query(c, k, select * from table) in
11           decrypt(k, data)
12   } in let result =  #(records customers) in
13       display(result);
14       let balance_customer = choose_customer(result)
15       let socket = connect(server1) in
16         write(socket, balance_customer)
```

The core of the snippet above is the behavioural variation (lines 2–11) bound to `records` that downloads the table of customers. The behavioural variation is a construct similar to pattern matching where goals replace patterns and whose execution triggers a dispatching mechanism. In our case, there are two alternatives which depend on the location and on the capabilities of the current user. 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. As an example, when outside the office the IP address of an available proxy is retrieved by the predicate `proxy` that binds the 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 (the API function `get_db` at line 9) in order to obtain the required data. Note that the third argument of the call to `crypto_query` 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 occur in the snippet above: the database connection `c` at line 4, a cryptographic key `k` at line 7, the address and a connection to the `server1` at line 15. Other API functions are: `open_db` at line 4, `query` at line 5, `decrypt` at line 11 and `write` at line 16. (Note that at line 14 we assume that the function `choose_customer` interactively asks the name of the customer to the user.)

To dynamically update the context, we use the constructs `tell` and `retract`, that add and remove Datalog facts, respectively. For example, the following code transfers the right to access the database `DB2` from Jane to Bob:

```
retract has_auth(Jane,DB2)
tell has_auth(Bob,DB2)
```

An application fails to adapt to a context (*functional failure*), when the dispatching mechanism fails. Another kind of failure happens when an application does not manipulate resources as expected (*non-functional failure*).

As an example of non-functional failure, assume that the company at a certain point decides to protect data about its customers. To do that, it constraints a vendor's application to open no further connections once connected to the company proxy, when out of the office — inside, a firewall is assumed to do the job. The application above violates this constraint because it computes the balance of a customer and sends it to `server1`.

Our two-phase static analysis prevents programs from experiencing either kind of failures. It consists of a type and effect system at compile time and of a control flow analysis at loading time.

The compilation results in a triple (C_p, e, H) made of the application context, the object code and an effect. Types are (almost) standard, and H is an over-approximation, called *history expression*, of the actual runtime behaviour of e. The effect abstractly represents the changes and the queries performed on the context and the invocations to the API functions at runtime.

For example, the type of function `main` is `unit` $\rightarrow$ `unit`, and the history expression of the fragment between lines 6 and 11 is

$$H = ask\,G \cdot connect(address)\langle H_c\rangle \cdot get\_db(channel)\langle H_g\rangle \cdot \\ crypto\_query(database)\langle H_q\rangle \cdot decrypt(key)\langle H_k\rangle$$

where $ask\,G$ represents (a call to the Datalog deduction machinery on) the goal in lines 6 and 7, followed by four abstract calls, corresponding to the API invocations in lines 8–11 ($\cdot$ stands for sequential composition). The abstract calls have the form $f(k)\langle H\rangle$, where k is the kind of the resource affected by f and the history expression H is its latent effect as declared by the API. For example, $get\_db(channel)\langle H_g\rangle$ corresponds to the invocation at line 9, and indicates that the resource is a channel, and that H_g is the latent effect of the system function `get_db`. Note that the function f changes the resource state *and* the context, e.g. through a `tell/retract`. Consequently, the latent effect registers these modifications. Indeed, most likely H_g will contain an element $tell\ connected(Jane, DB2)$ to record in the context that the system function connected the current user to the selected database, say Jane to `DB2`.

At loading time, the virtual machine of ML_{CoDa} performs two steps: linking and verification. The first step links the API to its actual implementation, and it constructs the initial context C, by combining the one of the application C_p with the system context that includes information on the actual state of the system, e.g. available resources and their usage constraints. Of course, the context C is checked for consistency. Then our last mile verification begins: it checks whether no functional failure occurs, i.e. whether the application adapts to all evolutions of C that may occur at runtime. And then it checks non-functional failures, i.e. whether resources are used in accordance with the rules established by the system that loads the program. Only those which pass the verification will be run. To do that conveniently and efficiently, we build a graph $\mathcal{G}$ describing the

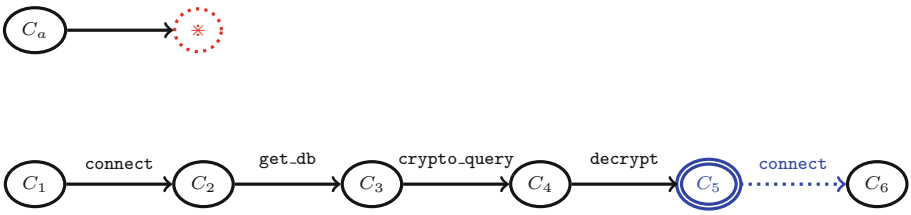

Fig. 1. Two evolution graphs showing a functional failure (top) and a non-functional failure (bottom) (Colour figure online)

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 context arising at runtime and its edges are labelled by the action which carried out the context change. 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.

The example above is rather simple, but suffices to show a functional and a non-functional failure. For the first, consider Alan who runs the application above. The graph shown in Fig. 1 (top) results from our analysis, where C_a is the initial context. Since he is not authorised to access the company database, the behavioural variation `records` fails (the predicate `has_auth` is false in C_a). The failure is shown in the graph of Fig. 1 because the failure node (dotted and in red in the pdf) $\lightning$ is reachable.

A non-functional failure occurs when Jane runs the application outside the office. The initial context now is different from C_a, and the graph in Fig. 1 (bottom) displays how this context evolves when the API calls in the code are carried out. Since Jane is outside, the second case of the behavioural variation `records` is selected. The application violates the constraint informally introduced above (once connected from outside, no other connections are allowed), because the function `connect` attempts to establish a new connection to `server1` at line 15 (represented by the dotted edge and the node, drawn in blue in the pdf).

3 $\mathbf{ML_{CoDa}}$ with Resources

We briefly define the syntax and the operational semantics of our extension of $\mathrm{ML_{CoDa}}$ to explicitly deal with resources; we mainly concentrate on its peculiar constructs, as those inherited by Datalog and ML are standard.

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

The Datalog part is standard: a program is a set of facts and clauses. We assume that each program is safe [7]; to deal with negation, we adopt *Stratified Datalog* under the Closed World Assumption.

The functional part inherits most of the ML constructs. Besides the usual ones, our values include Datalog facts F, behavioural variations and resource handles r. Also, we introduce the set $\tilde{x} \in DynVar$ of *parameters*, i.e. variables assuming values depending on the properties of the running context, while $x, y \in Var$ are standard identifiers, disjoint from parameters. Our COP constructs 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 possibly free in e_i). At runtime, the first goal G_i satisfied by the context determines the expression e_i to be selected (*dispatching*). 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. The append operator $e_1 \cup e_2$ concatenates behavioural variations, so allowing for dynamic composition. The application 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, if any. We assume that the programmer can invoke a set of functions provided by the API, by writing $f(e_1, \ldots, e_n)$. The syntax follows:

$$
\begin{aligned}
&\tilde{x} \in DynVar\,(Var \cap DynVar = \emptyset) \quad C, C_p \in Context \quad r \in Res \quad f \in API \\
&Va ::= G.e \mid G.e, Va \\
&v ::= c \mid \lambda_y x.e \mid (x)\{Va\} \mid F \mid r \\
&e ::= v \mid x \mid \tilde{x} \mid e_1\, e_2 \mid let\, x\, =\, e_1\, in\, e_2 \mid if\, e_1\, then\, e_2\, else\, e_3 \mid \\
&\qquad dlet\, \tilde{x}\, =\, e_1\, when\, G\, in\, e_2 \mid tell(e_1) \mid retract(e_1) \mid e_1 \cup e_2 \mid \#(e_1, e_2) \mid \\
&\qquad f(e_1, \ldots, e_n)
\end{aligned}
$$

Semantics. For the Datalog evaluation, we adopt the top-down standard semantics for stratified programs [7]. Given a context $C \in Context$ and a goal G, $C \vDash G\, with\, \theta$ means that the goal G, under the substitution θ replacing constants for variables, is satisfied in the context C.

The small-step operational semantics of ML_{CoDa} is defined for expressions with no free variables, but possibly with free parameters, allowing for openness. For that, we have an environment $\rho\colon DynVar \to Va$, mapping parameters to variations. 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'. We assume that the initial configuration is $\rho_0 \vdash C, e_p$ where ρ_0 contains the bindings for all system parameters, and C results from linking the system and the application contexts.

Figure 2 shows the inductive definitions of the reduction rules for the constructs typical of ML_{CoDa}; the other ones are standard, and such are the congruence rules that reduce subexpressions, e.g. $\rho \vdash C, tell(e) \to C', tell(e')$ if $\rho \vdash C, e \to C', e'$. See [11] for full definitions. Below, we briefly comment on the rules displayed.

The rule for $tell(e)/retract(e)$ evaluates the expression e until it reduces to a fact F, which is a value of ML_{CoDa}. Then, the evaluation yields the unit value $()$ and a new context C', obtained from C by adding/removing F. The following

$$\frac{\rho \vdash C,\ e \to C',\ e'}{\rho \vdash C,\ tell(e) \to C',\ tell(e')}\ (\textsc{Tell1}) \qquad \frac{}{\rho \vdash C,\ tell(F) \to C \cup \{F\},\ ()}\ (\textsc{Tell2})$$

$$\frac{\rho \vdash C,\ e \to C',\ e'}{\rho \vdash C,\ retract(e) \to C',\ retract(e')}\ (\textsc{Retract1}) \qquad \frac{}{\rho \vdash C,\ retract(F) \to C \backslash \{F\},\ ()}\ (\textsc{Retract2})$$

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

$$\frac{}{\rho \vdash C,\ dlet\ \tilde{x} = e_1\ when\ G\ in\ v \to C,\ v}\ (\textsc{Dlet2}) \qquad \frac{\rho(\tilde{x}) = Va \quad dsp(C,\ Va) = (e,\ \{\overline{c}\ /\ \overrightarrow{y}\})^{\rightarrow}}{\rho \vdash C,\ \tilde{x} \to C,\ e\{\overline{c}\ /\ \overrightarrow{y}\}^{\rightarrow}}\ (\textsc{Par})$$

$$\frac{\rho \vdash C,\ e_1 \to C',\ e_1'}{\rho \vdash C,\ e_1 \cup e_2 \to C',\ e_1' \cup e_2}\ (\textsc{Append1}) \qquad \frac{\rho \vdash C,\ e_2 \to C',\ e_2'}{\rho \vdash C,\ (x)\{Va_1\} \cup e_2 \to C',\ (x)\{Va_1\} \cup e_2'}\ (\textsc{Append2})$$

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

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

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

$$\frac{\rho \vdash C, e_i \to C', e_i'}{\rho \vdash C,\ f(v_1, \ldots, e_i, \ldots, e_n) \to C', f(v_1, \ldots, e_i', \ldots, e_n)}\ (\textsc{Res1}) \qquad \frac{v = syscall(f, r, v_2, \ldots, v_n)}{\rho \vdash C,\ f(r, v_2, \ldots, v_n) \to C', v}\ (\textsc{Res2})$$

Fig. 2. The reduction rules for the constructs peculiar of $\mathrm{ML_{CoDa}}$

example shows the reduction of a *tell* construct, where we apply the function $\texttt{foo} = \lambda \texttt{x}.\ \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{F}_2\ \texttt{else}\ \texttt{F}_3$ to unit, assuming that e_1 reduces to false without changing the context:

$$\rho \vdash C,\ \texttt{tell}(\texttt{foo}\,()) \to^* C,\ \texttt{tell}(\texttt{F}_3) \to C \cup \{\texttt{F}_3\},\ ()$$

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

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 by the following partial function:

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

A variation is inspected from left to right to find the first goal G satisfied by the context C ($C \models G$), under a substitution θ. If this search succeeds, the dispatching returns the corresponding expression e and θ. Then $\tilde{x}$ reduces to $e\,\theta$. Instead, if the dispatching fails because no goal holds, the computation gets stuck since the program cannot adapt to the current context.

As an example of context-dependent binding consider the expression $\texttt{tell}(\tilde{x})$, in an environment ρ that binds the parameter $\tilde{x}$ to $e' = \texttt{G}_1.\texttt{F}_5, \texttt{G}_2.\texttt{foo}\,()$ (`foo` is defined above) and in a context C that satisfies the goal G_2 but not G_1:

$$\rho \vdash C,\, \texttt{tell}(\tilde{x}) \;\rightarrow\; C,\, \texttt{tell}(\texttt{foo}\,()) \;\rightarrow^*\; C,\, \texttt{tell}(\texttt{F}_3) \;\rightarrow\; C \cup \{\texttt{F}_3\}, ()$$

In the first step, we retrieve the binding for $\tilde{\texttt{x}}$ (recall it is e'), where $dsp(C,\, e') = dsp(C, \texttt{G}_1.\texttt{F}_5, \texttt{G}_2.\texttt{foo}\,()) = (\texttt{foo}\,(), \theta)$, for a suitable substitution θ.

The rules for $e_1 \cup e_2$ sequentially evaluate e_1 and e_2 until they reduce to behavioural variations. Then, they are concatenated (bound variables are renamed to avoid name captures, see rule (Append3)). As an example of concatenation, let `T` be the goal always true, and consider the function $doo = \lambda x.\lambda y.\, x \cup (w)\{\texttt{T}.y\}$. It takes as arguments a behavioural variation x and a value y, and it extends x by adding a default case which is always selected when no other case applies. In the following computation we apply doo to the behavioural variation $bv = (x)\{G_1.c_1, G_2.x\}$ and to c_2 (c_1, c_2 constants):

$$\rho \vdash C,\, doo\,p\,c_2 \;\rightarrow\; C,\, (x)\{G_1.c_1, G_2.x\} \cup (w)\{T.c_2\} \;\rightarrow\; C,\, (z)\{G_1.c_1, G_2.z, T.c_2\}$$

The behavioural variation application $\#(e_1, e_2)$ 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 replaced x. Also in this case the computation gets stuck if the dispatching mechanism fails. As an example, consider the above behavioural variation bv and apply it to the constant c in a context C that satisfies the goal G_2 but not G_1. Since $dsp(C,\, bv) = dsp(C, (x)\{G_1.c_1, G_2.x\}) = (x,\, \theta)$ for some substitution θ, we get

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

The rules for API invocation first evaluate the arguments, and then run the code of f through the meta function *syscall*, possibly affecting the context. For simplicity, we assume that a single resource handle occurs in an API invocation, as its first argument.

4 Types

4.1 History Expressions

History Expressions [3] are a basic process algebra used to soundly abstract the set of execution histories that a program may generate. Here, history expressions approximate the sequence of actions that a program may perform over the

$$\frac{}{C,\ \epsilon \cdot H \to C,\ H} \qquad \frac{}{C, \mu h.H \to C, H[\mu h.H/h]} \qquad \frac{C, H_1 \to C',\ H_1'}{C, H_1 + H_2 \to C',\ H_i'}$$

$$\frac{C, H_2 \to C',\ H_2'}{C, H_1 + H_2 \to C',\ H_2'} \qquad \frac{C,\ H_1 \to C',\ H_1'}{C,\ H_1 \cdot H_2 \to C',\ H_1' \cdot H_2} \qquad \frac{}{C,\ tell\, F \to C \cup \{F\},\ \epsilon}$$

$$\frac{}{C,\ retract\, F \to C \backslash \{F\},\ \epsilon} \qquad \frac{C, H \to C',\ H'}{C, f(k)\langle H \rangle \to C',\ f(k)\langle H' \rangle} \qquad \frac{}{C, f(k)\langle \epsilon \rangle \to C,\ \epsilon}$$

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

Fig. 3. Semantics of History Expressions

context at runtime, i.e. asserting/retracting facts and asking if a goal holds, as well as how behavioural variations will be "resolved". In addition, we record a call to an API function, together with its abstract behaviour, represented as a history expression.

The syntax of history expressions is the following

$$H ::= \epsilon \mid h \mid \mu h.H \mid H_1 + H_2 \mid H_1 \cdot H_2 \mid tell\, F \mid retract\, F \mid f(k)\langle H \rangle \mid \Delta$$
$$\Delta ::= ask\, G.H \otimes \Delta \mid fail$$

The empty history expression ϵ abstracts programs which do not interact with the context; $\mu h.H$ represents possibly recursive functions, where h is the recursion variable; the non-deterministic sum $H_1 + H_2$ stands for the conditional expression *if-then-else*; the concatenation $H_1 \cdot H_2$ is for sequences of actions that arise, e.g. while evaluating applications; the "atomic" history expressions $tell\, F$ and $retract\, F$ are for the analogous expressions of ML_{CoDa}; the history expression for an API invocation is rendered by $f(k)\langle H \rangle$, where f acts on a resource of type k, and H is the history expression declared in the API; Δ is an *abstract variation*, defined as a list of history expressions, each element H_i of which is guarded by an $ask\, G_i$, so to mimic our dispatching mechanism. For an example of abstract variation, see the history expression H in Sect. 2.

Given a context C, the behaviour of a closed history expression H (i.e. with no free variables) is formalised by the transition system inductively defined in Fig. 3. 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 in [3], and below we briefly comment on them.

The recursion $\mu h.H$ reduces to its body H substituting $\mu h.H$ for the recursion variable h. The sum $H_1 + H_2$ non-deterministically reduces to the history expression obtained by reducing either H_1 or H_2. The sequence $H_1 \cdot H_2$ reduces to H_2, provided that H_1 step-wise becomes ϵ. An action $tell\, F$ reduces to ϵ and yields a context C' where the fact F has just been added; similarly for $retract\, F$. The rules for an API invocation evaluate the body H until termination.

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 overall 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 C, fail is reached, representing that the dispatching mechanism fails.

4.2 Types and Effects

We extend in Figs. 5 and 4 the logical presentation of a type and effect system for $\mathrm{ML}_{\mathrm{CoDa}}$ of [9] by introducing a family of types $res(k)$ for every kind k of resource. As done there, we assume a Datalog typing function γ that given a goal G returns a list of pairs $(x, \text{type-of-}x)$, for all the variables x of G (γ is used e.g. in the rule TVARIATION in Fig. 5).
The syntax of types is

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

We have types for constants (*int*, *bool*, *unit*, ...), resource types, functional types, behavioural variations types, and facts. Some types are annotated to support our static analysis. In the type $fact_\phi$ the set ϕ soundly contains the facts that an expression can be reduced to at runtime (see the rules of the semantics (TELL2) and (RETRACT2)). In the type $\tau_1 \xrightarrow{K|H} \tau_2$ associated with a function f, the environment K is a precondition needed to apply f. The environment K maps a parameter $\tilde{x}$ to a pair consisting of a type and an abstract variation Δ, used to resolve the binding for $\tilde{x}$ at runtime, formally $K ::= \emptyset \mid K, (\tilde{x}, \tau, \Delta)$. As an annotation, K stores the types and the abstract variations of parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e. the sequence of actions that may be performed over the context during the function evaluation. Analogously, in the type $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv and Δ is an abstract variation representing the information that the dispatching mechanism uses at runtime to apply bv.

We now introduce the orderings $\sqsubseteq_H, \sqsubseteq_\Delta, \sqsubseteq_K$ on H, Δ and K, respectively (often omitting the indexes when unambiguous). We define $H_1 \sqsubseteq H_2$ iff $\exists H_3$ such that $H_2 = H_1 + H_3$; $\Delta_1 \sqsubseteq \Delta_2$ iff $\exists \Delta_3$ such that $\Delta_2 = \Delta_1 \otimes \Delta_3$, (note that we assume $\mathit{fail} \otimes \Delta = \Delta$, so Δ has a single trailing term *fail*); $K_1 \sqsubseteq K_2$ iff $(\,(\tilde{x}, \tau_1, \Delta_1) \in K_1$ implies $(\tilde{x}, \tau_2, \Delta_2) \in K_2 \wedge \tau_1 \leq \tau_2 \wedge \Delta_1 \sqsubseteq \Delta_2\,)$.

Typing judgements $\Gamma; K \vdash e : \tau \triangleright H$ mean that in the standard type environment Γ and in the *parameter environment* K, the expression e has type τ and effect H. Furthermore, we assume that the type of every API function f is stored in the typing environment Γ, i.e. $\Gamma(f) = res(k) \times \tau_2 \times \cdots \times \tau_n \xrightarrow{\epsilon;H} \tau_b$, where, by abuse of notation, we use a tuple type for the domain of f (see the rule (TRES) below).

$$\text{(Stconst)}\quad \tau_b \leq \tau_b \qquad\qquad \text{(Sfact)}\quad \frac{\phi \subseteq \phi'}{fact_\phi \leq fact_{\phi'}}$$

$$\text{(Sfun)}\quad \frac{\tau_1' \leq \tau_1 \quad K \sqsubseteq K' \qquad \tau_2 \leq \tau_2' \quad H \sqsubseteq H'}{\tau_1 \xrightarrow{K|H} \tau_2 \leq \tau_1' \xrightarrow{K'|H'} \tau_2'} \qquad \text{(Sva)}\quad \frac{\tau_1' \leq \tau_1 \quad K \sqsubseteq K' \qquad \tau_2 \leq \tau_2' \quad \Delta \sqsubseteq \Delta'}{\tau_1 \xRightarrow{K|\Delta} \tau_2 \leq \tau_1' \xRightarrow{K'|\Delta'} \tau_2'}$$

Fig. 4. The subtyping relation

We now briefly comment on the most interesting rules; more comments and examples can be found in [9]. As expected the rules for subtyping and subeffecting (Fig. 4) say that the subtyping relation is reflexive (rule (Srefl)); that a type $fact_\phi$ is a subtype of a type $fact_{\phi'}$ whenever $\phi \subseteq \phi'$ (rule (Sfact)); that functional types are contra-variant in the types of arguments and covariant in the type of the result and in the annotations (rule (Sfun)); analogously for the types of behavioural variations (rule (Sva)).

The rule (Tsub) allows us to freely enlarge types and effects by applying the subtyping and subeffecting rules. The rule (Tfact) says that a fact F has type *fact* annotated with the singleton $\{F\}$ and empty effect. The rule (Ttell)/(Tretract) asserts that the expression $tell(e)/retract(e)$ has type *unit*, provided that the type of e is $fact_\phi$. The overall effect is obtained by concatenating the effect of e with the non-deterministic summation of $tell\,F/retract\,F$ where F is any of the facts in the type of e. Rule (Tpar) looks for the type and the effect of the parameter $\tilde{x}$ in the environment K. In the rule (Tvariation) we guess an environment K' and the type τ_1 for the bound variable x. We determine the type for each subexpression e_i under K' and the environment Γ extended by the type of x and of the variables $\overrightarrow{y_i}$ occurring in the goal G_i (recall that the Datalog typing function γ returns a list of pairs (z, type-of-z) for all variable z of G_i). Note that all subexpressions e_i have the same type τ_2. We also require that the abstract variation Δ results from concatenating $ask\,G_i$ with the effect computed for e_i. The type of the behavioural variation is annotated by K' and Δ. The rule (Tvapp) type-checks behavioural variation applications and reveals the role of preconditions. As expected, e_1 is a behavioural variation with parameter of type τ_1 and e_2 has type τ_1. We get a type if the environment K', which acts as a precondition, is included in K according to $\sqsubseteq$. The type of the behavioural variation application is τ_2, i.e. the type of the result of e_1, and the effect is obtained by concatenating the ones of e_1 and e_2 with the history expression Δ, occurring in the annotation of the type of e_1. The rule (Tappend) asserts that two expressions e_1,e_2 with the same type τ, except for the abstract variations Δ_1, Δ_2 in their annotations, and effects H_1 and H_2, are combined into $e_1 \cup e_2$ with type τ, and concatenated annotations and effects. More precisely, the resulting annotation has the same precondition of e_1 and e_2 and abstract

$$\text{(TSUB)}\ \frac{\Gamma;\, K \vdash e \,:\, \tau' \triangleright H' \qquad \tau' \leq \tau \qquad H' \sqsubseteq H}{\Gamma;\, K \vdash e \,:\, \tau \triangleright H} \qquad \text{(TCONST)}\ \frac{}{\Gamma;\, K \vdash c \,:\, \tau_c \triangleright \epsilon}$$

$$\text{(TFACT)}\ \frac{}{\Gamma;\, K \vdash F \,:\, fact_{\{F\}} \triangleright \epsilon} \qquad \text{(TVAR)}\ \frac{\Gamma(x) = \tau}{\Gamma;\, K \vdash x \,:\, \tau \triangleright \epsilon}$$

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

$$\text{(TLET)}\ \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1 \qquad \Gamma; x : \tau_1,\, K \vdash e_2 \,:\, \tau_2 \triangleright H_2}{\Gamma;\, K \vdash let\, x \,=\, e_1\, in\, e_2 \,:\, \tau_2 \triangleright H_1 \cdot H_2} \qquad \text{(TTELL)}\ \frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H}{\Gamma;\, K \vdash tell(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} tell\, F\right)}$$

$$\text{(TRETRACT)}\ \frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H}{\Gamma;\, K \vdash retract(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} retract\, F\right)} \qquad \text{(TABS)}\ \frac{\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K' \vdash e \,:\, \tau_2 \triangleright H}{\Gamma;\, K \vdash \lambda_f x.e \,:\, \tau_1 \xrightarrow{K'|H} \tau_2 \triangleright \epsilon}$$

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

$$\text{(TAPPEND)}\ \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|\Delta_1} \tau_2 \triangleright H_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \xrightarrow{K'|\Delta_2} \tau_2 \triangleright H_2}{\Gamma;\, K \vdash e_1 \cup e_2 \,:\, \tau_1 \xrightarrow{K'|\Delta_1 \otimes \Delta_2} \tau_2 \triangleright H_1 \cdot H_2}$$

$$\text{(TVAPP)}\ \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright H_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2 \qquad K' \sqsubseteq K}{\Gamma;\, K \vdash \#(e_1, e_2) \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot \Delta}$$

$$\text{(TAPP)}\ \frac{\Gamma;\, K \vdash e_1 \,:\, \tau_1 \xrightarrow{K'|H_3} \tau_2 \triangleright H_1 \qquad \Gamma;\, K \vdash e_2 \,:\, \tau_1 \triangleright H_2 \qquad K' \sqsubseteq K}{\Gamma;\, K \vdash e_1\, e_2 \,:\, \tau_2 \triangleright H_1 \cdot H_2 \cdot H_3} \qquad \text{(TPAR)}\ \frac{K(\tilde{x}) = (\tau,\, \Delta)}{\Gamma;\, K \vdash \tilde{x} \,:\, \tau \triangleright \Delta}$$

$$\text{(TDLET)}\ \frac{\Gamma, \overline{y} : \overrightarrow{\tau}; \overrightarrow{K} \vdash e_1 \,:\, \tau_1 \triangleright H_1 \qquad \Gamma;\, K, (\tilde{x},\, \tau_1,\, \Delta') \vdash e_2 \,:\, \tau_2 \triangleright H_2}{\Gamma;\, K \vdash dlet\, \tilde{x} \,=\, e_1\, when\, G\, in\, e_2 \,:\, \tau_2 \triangleright H_2} \quad \begin{array}{l} \text{where } \gamma(G) = \overline{y} : \overrightarrow{\tau} \quad \rightarrow \\ \Delta' = \begin{cases} (G.H_1 \otimes \Delta) & \text{if } K(\tilde{x}) = (\tau_1,\, \Delta) \\ (G.H_1 \otimes fail) & \text{if } \tilde{x} \notin K \end{cases} \end{array}$$

$$\text{(TRES)}\ \frac{\Gamma(f) : res(k) \times \tau_2 \times \cdots \times \tau_n \xrightarrow{\emptyset;H} \tau_b \qquad \Gamma; K \vdash e_i : \tau_i \triangleright H_i}{f(e_1, \ldots, e_n) : \tau_b \triangleright H_1 \cdot \ldots \cdot H_n \cdot f(k)\langle H \rangle} \quad \text{where } \tau_1 = res(k)$$

Fig. 5. Type and effect system

variation $\Delta_1 \otimes \Delta_2$, and effect $H_1 \cdot H_2$. 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 information on the parameter $\tilde{x}$. The type and the effect for the overall *dlet* expression are those of e_2. Finally, the rule (TRES) retrieves the type of f from Γ and type-checks its

arguments. The resulting type is the retrieved one for f and the overall effect is the concatenation of the effects of the arguments and $f(k)\langle H\rangle$, where k denotes the kind of the resource affected and H is the latent effect of f. For simplicity we assume that f manipulates a single resource occurring in first position and that f can always be applied so its type has no preconditions.

As an example, consider the behavioural variation $bv_1 = (x)\{G_1.f(e_1), G_2.e_2\}$. Let Γ' be the environment $\Gamma, x : int, f : res(k_1) \to \tau$ (goals have no variables) and K' be the parameter environment. Then assume that under these environments e_1 has type $res(k_1)$ and effect H_r, and that the two cases of this behavioural variation have type τ and effects $H_1 = f(k_1)\langle H_r\rangle$ and H_2, respectively. 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.

Our type and effect system is sound with respect to the operational semantics of $\mathrm{ML}_{\mathrm{CoDa}}$. To concisely state soundness, it is convenient to introduce the following technical definition and to exploit the following results.

Definition 1 (Type of dynamic environment). *Given the type and parameter 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 $\forall \tilde{x} \in dom(\rho)$ such that $\rho(\tilde{x}) = G_1.e_1, \ldots, G_n.e_n$ and $K(\tilde{x}) = (\tau, \Delta)$, $\forall i \in [1, n]$ the following hold:*
(a) $\gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ and (b) $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K \vdash e_i : \tau' \triangleright H_i$ and (c) $\tau' \leq \tau$ and (d) $\bigotimes_{i\in[1,n]} G_i.H_i \sqsubseteq \Delta$.

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 $\Gamma; K \vdash e_s : \tau \triangleright H_s$ and $\rho \vdash C, e_s \to C', e'_s$ then $\Gamma; K \vdash e'_s : \tau \triangleright H'_s$ and $C, H_s \to^* C', H$ for some $H \sqsubseteq H'_s$.*

This theorem is quite standard: types are preserved under computations and the effect statically determined includes the one reached by the considered computation. However, the Progress Theorem assumes that the effect H does not reach *fail*, i.e. that the dispatching mechanism succeeds at runtime. We take care of ensuring this property in Sect. 5 (below $\rho \vdash C, e \nrightarrow$ means that there exist no C' and e' such that $\rho \vdash C, e \to C', e'$). The following corollary ensures that the history expression obtained as an effect of e over-approximates the actions that may be performed over the context during the evaluation of e.

Theorem 2 (Progress). *Let e_s be a closed expression s.t. $\Gamma; K \vdash e_s : \tau \triangleright H_s$; and let ρ be a dynamic environment s.t. $dom(\rho)$ includes the set of parameters of e_s, and $\Gamma \vdash \rho : K$. If $\rho \vdash C, e_s \nrightarrow \wedge\ C, H_s \nrightarrow^+ C', fail$ then e_s is a value.*

Corollary 1 (Over-approximation). *Let e_s be a closed expression. If $\Gamma; K \vdash e_s : \tau \triangleright H_s \ \wedge\ \rho \vdash C, e_s \to^* C', e'$, for some ρ such that $\Gamma \vdash \rho : K$, then $\Gamma; K \vdash e' : \tau \triangleright H'_s$ and there exists a sequence of transitions $C, H_s \to^* C', H'$ for some $H' \sqsubseteq H'_s$.*

The following theorem ensures the correctness of our approach.

Theorem 3 (Correctness). *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s$; 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 $C, H_s \twoheadrightarrow^+ C'$, 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.*

5 Loading Time Analysis

As anticipated in Sect. 1, the ML_{CoDa} compiler produces a triple (C_p, e_p, H_p) made of the application context C_p, the object code e_p, and an effect H_p over-approximating the behaviour of the application. Using it, the virtual machine of ML_{CoDa} performs a linking and a verification phase at loading time. During the linking phase, system variables are resolved and the initial context C is constructed, combining C_p and the system context, provided that the result is consistent. Still, the application is "open" with respect to its parameters. This calls for the last mile verification phase: we check whether the application adapts to all the evolutions of C that may occur at runtime, i.e., that all dispatching invocations will always succeed. And then we check that resources are used in accordance with the rules established by the system loading the program. Only programs which pass this verification phase will be run. To do that conveniently and efficiently, we build a graph $\mathcal{G}$ describing all the possible evolutions of the initial context, exploiting the history expression H_p. Technically, we compute $\mathcal{G}$ through a static analysis of history expressions with a notion of validity; intuitively, a history expression is valid for an initial context if the dispatching mechanism always succeeds. Our static analysis is specified in terms of Flow Logic [14,16], a declarative approach borrowing from and integrating many classical static techniques. Flow Logic has been applied to a wide variety of programming languages and calculi of computation including calculi with functional, imperative, object-oriented, concurrent, distributed, and mobile features, among many see [4,6,10,12,15].

To support the formal development, we assume that history expressions are mechanically labelled from a given set $Lab = Lab_H \uplus Lab_S$, with typical element l. The elements of Lab_H label the abstract counterparts of ML_{CoDa} constructs, while those of Lab_S occur in the declared latent effects of the API functions (sometimes, with $\hat{l}$ as typical element). Formally:

$$
\begin{aligned}
H ::= {} & \mathfrak{d} \mid \epsilon^l \mid h^l \mid (\mu h.H)^l \mid tell\, F^l \mid retract\, F^l \mid f(k)^l \langle H^{\hat{l}} \rangle \mid \\
& (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \Delta \\
\Delta ::= {} & (ask\, G.H \otimes \Delta)^l \mid fail^l
\end{aligned}
$$

For technical reasons, we introduce a new empty history expression $\mathfrak{d}$ which is unlabelled. This is because our analysis is syntax-driven, and we need to distinguish when the empty history expression comes from the syntax (ϵ^l) and

when it is instead obtained by reduction in the semantics ($\mathfrak{d}$). The semantics of history expressions is accordingly modified, by always allowing the transition $C, \epsilon^l \rightarrow C, \mathfrak{d}$. Furthermore, w.l.o.g. we assume that all the bound variables occurring in a history expression are distinct. To keep track of a bound variable h^l introduced in $(\mu h.H_1^{l_1})^{l_2}$, we shall use a suitable function $\mathbb{K}$.

The static approximation is represented by an *estimate* $(\Sigma_\circ, \Sigma_\bullet)$, given by the pair of functions $\Sigma_\circ, \Sigma_\bullet : Lab \rightarrow \wp(Context \cup \{\divideontimes\})$, where $\divideontimes$ is the distinguished "failure" context representing a dispatching failure. For each label l, the *pre-set* $\Sigma_\circ(l)$ and the *post-set* $\Sigma_\bullet(l)$ over-approximate the set of contexts possibly arising *before* and *after* the evaluation of H^l, respectively.

We inductively specify our analysis in Fig. 6 by defining the validity relation

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

where $\mathcal{AE} = (Lab \rightarrow \wp(Context \cup \{\divideontimes\}))^2$ is the domain of the results of the analysis and $\mathbb{H}$ the set of history expressions. We write $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, when the pair $(\Sigma_\circ, \Sigma_\bullet)$ is an acceptable analysis estimate for the history expression H^l. The notion of acceptability will then be used in Definition 3 to check whether H, hence the expression e it is an abstraction of, will never fail in a given initial context C. Below, we briefly comment on the inference rules, where $\mathcal{E} = (\Sigma_\circ, \Sigma_\bullet)$ and immaterial labels are omitted.

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

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

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

$$\frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1} \qquad (\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\bullet(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (H_1^{l_1} \cdot H_2^{l_2})^l} \text{(Aseq1)}$$

$$\frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\bullet(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (\mathfrak{d} \cdot H_2^{l_2})^l} \text{(Aseq2)}$$

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

$$\frac{\forall C \in \Sigma_\circ(l) \qquad (C \vDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)) \qquad (C \nvDash G \implies (\Sigma_\circ, \Sigma_\bullet) \vDash \Delta^{l_2} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_2) \qquad \Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l))}{(\Sigma_\circ, \Sigma_\bullet) \vDash (askG.H^{l_1} \otimes \Delta^{l_2})^l} \text{(Aask1)}$$

$$\frac{\divideontimes \in \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash fail^l} \text{(Aask2)} \qquad \frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \qquad \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash (\mu h.H^{l_1})^l} \text{(Arec)}$$

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

$$\frac{(\Sigma_\circ, \Sigma_\bullet) \vDash H^{\hat{l}} \qquad \Sigma_\circ(l) \subseteq \Sigma_\circ(\hat{l}) \qquad \Sigma_\bullet(\hat{l}) \subseteq \Sigma_\bullet(l)}{(\Sigma_\circ, \Sigma_\bullet) \vDash f(k)^l \langle H^{\hat{l}} \rangle} \text{(Ares)}$$

Fig. 6. Specification of the analysis for History Expressions

the abstraction $f(k)^l\langle H^{\hat{l}}\rangle$ of an API function. It requires that $(\Sigma_\circ, \Sigma_\bullet)$ is an acceptable estimate for $H^{\hat{l}}$ and that the pre-set of $\hat{l}$ includes that of l, while the inverse relation holds for the post-sets.

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

Definition 2 (Valid analysis estimate). *Given H^l and an initial context C, we say that a pair $(\Sigma_\circ, \Sigma_\bullet)$ is a valid analysis estimate for H and C iff $C \in \Sigma_\circ(l_p)$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$.*

The set of estimates can be partially ordered in the standard way, and shown to form a Moore family. Therefore, there always exists a minimal valid analysis estimate [16] (see [9] Th. 4). The correctness of our analysis follows from subject reduction.

Theorem 4 (Subject Reduction). *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$. If for all $C \in \Sigma_\circ(l)$ such that $C, H^l \rightarrow C', H''^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H''^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.*

Now we can define when a history expression H_p is viable for an initial context C, i.e. when it passes the verification phase. In the following definition, let $lfail(H)$ be the set of labels of the $fail$ sub-terms in H:

Definition 3 (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 $\forall l \in dom(\Sigma_\bullet) \backslash lfail(H_p)$, $\divideontimes \notin \Sigma_\bullet(l)$.*

Table 1. An estimate for the history expression H_a in the context $C = \{F_2, F_5\}$.

	$\Sigma^1_\circ$	$\Sigma^1_\bullet$
1	$\{\{F_2, F_5\}\}$	$\{\{F_1, F_2, F_5\}\}$
2	$\{\{F_1, F_2, F_5\}\}$	$\{\{F_1, F_5\}\}$
3	$\{\{F_1, F_5\}\}$	$\{\{F_1\}\}$
4	$\{\{F_1, F_5\}\}$	$\{\{F_1\}\}$
5	$\{\{F_1, F_2, F_5\}\}$	$\{\{F_1\}\}$
6	$\{\{F_2, F_5\}\}$	$\{\{F_1\}\}$
7	$\{\{F_2, F_5\}\}$	$\{\{F_1, F_5\}\}$
8	$\{\{F_2, F_5\}\}$	$\{\{F_2, F_5, F_8\}\}$
9	$\{\{F_2, F_5, F_8\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
10	$\{\{F_2, F_5\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
11	$\emptyset$	$\emptyset$
12	$\emptyset$	$\emptyset$
13	$\emptyset$	$\emptyset$
14	$\{\{F_2, F_5\}\}$	$\{\{F_1, F_2, F_5, F_8\}\}$
15	$\{\{F_2, F_5\}\}$	$\{\{F_1\}, \{F_1, F_2, F_5, F_8\}\}$

As an example of viability checking, consider the context $C = \{F_2, F_5\}$, consisting of facts only, and the following history expression H_a:

$$H_a = ((tell\, F_1^1 \cdot (retract\, F_2^2 \cdot f(k_1)^3\langle retract\, F_5^4\rangle)^5)^6 + \\ g(k_2)^7\langle(ask\, F_5.(tell\, F_8^8 \cdot tell\, F_1^9)^{10} \otimes (ask\, F_3.retract\, F_4^{11} \otimes fail^{12})^{13})^{14}\rangle)^{15}$$

For each label l occurring in H_a, Table 1 shows the corresponding values of $\Sigma_\circ^1(l)$ and $\Sigma_\bullet^1(l)$, respectively.

Now we exploit the result of the above analysis to build the evolution graph $\mathcal{G}$, that describes how the initial context C evolves at runtime. The virtual machine can use $\mathcal{G}$ to predict how the application interacts with and affects the context and the resources.

In the following let $Fact^*$ and $Lab^* = Lab_H^* \uplus Lab_S^*$ be the set of facts and the set of labels occurring in H_p, the history expression under verification. Intuitively, $\mathcal{G}$ is a direct graph, the nodes of which are the set of contexts reachable from an initial context C, while running H_p. There is a labelled arc between two nodes C_1 and C_2 if C_2 is obtained from C_1 either through telling or removing a fact F, or through telling a set of facts and removing another set, when executing an API f. In the definition below the function $\mu : Lab_H^* \to \mathbb{H}$ recovers a construct in a given history expression $H \in \mathbb{H}$ from its label. Also let $\mathcal{A} = \{tell\, F^l, retract\, F^l, f(k)^l\langle H^{\hat{l}}\rangle \mid F \in Fact^* \wedge l, \hat{l} \in Lab^*\}$.

Definition 4 (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*

$$N = \bigcup_{l \in Lab_H^*} (\Sigma_\circ(l) \cup \Sigma_\bullet(l))$$
$$E = \{(C_1, C_2) \mid \exists l \in Lab_H^* \ s.t.\ \mu(l) \in \mathcal{A} \wedge C_1 \in \Sigma_\circ(l) \wedge (C_2 \in \Sigma_\bullet(l) \vee C_2 = \divideontimes)\}$$
$$L : E \to \mathcal{P}(\mathcal{A}) \ is\ such\ that\ \mu(l) \in L(t)\ iff\ t = (C_1, C_2) \in E \wedge C_1 \in \Sigma_\circ(l)$$

We can use the evolution graph $\mathcal{G}$ to verify that there are no functional or non-functional failures. The first case verifies viability, and simply consists in checking that the failure context $\divideontimes$ is not reachable from the initial one. The non-functional properties, a sort of CTL* formulae [1], constrain the usage of resources and predicate over nodes, i.e. contexts, and paths in the evolution graph. We can naturally check this kind of properties by visiting the graph.

Figure 7 depicts the evolution graph of the context C and the history expressions H_a introduced above. It is immediate checking that the node $\divideontimes$ is not reachable, thus showing in another way that H_a is viable for C. As an example of non-functional property, assume that the system requires that the program is not allowed to invoke the API function f on a resource of kind k_1 when the fact F_5 holds in the context. Verifying this property requires to visit the graph and to check that there is no arc labelled $f(k_1)$ from every node in which F_5 is true. We can easily detect that the node $\{F_1, F_5\}$ double circled (blue in the pdf) violates the requirement. One can also require a property on the context target of an API function. For instance, if the constraint is "after f the fact F_2 must hold" the target of f would be marked as a non-functional failure.

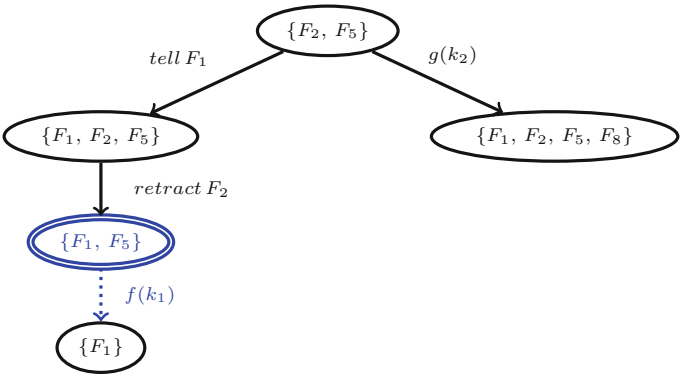

Fig. 7. The evolution graph for the context $C = \{F_2, F_5\}$ and the history expression H_a (only the nodes reachable from C are shown). (Color figure online)

6 Conclusions

We considered the problem of managing resources in adaptive systems. In these systems the context, i.e. the working environment, contains different kinds of resources and makes them available to applications, typically through specific handles. The actual capabilities of the available resources, their permitted usage and their number depend on the hosting system and are only known at runtime. To address these issues, we extended ML_{CoDa}, a two-component language for programming adaptive systems, with a notion of API providing programmers with a set of functions that allow them to manipulate resources.

When entering in a context an application can fail for two reasons: either because it is unable to adapt to the context or because it misuses a resource. To prevent this kind of non-functional failures to occur, we extended the type and effect system and the control flow analysis of [9]. Since parts of the context are unknown at compile time, the control flow analysis can only be carried out (on the effect) at loading time after the linking step. Indeed, full information about resources is only available when in the current context.

As future work we will study how to express the constraints over the usage of resources, e.g. in the form of CTL* formulas. A natural candidate approach for verifying that resources are correctly handled, would then be model-checking the evolution graph built at loading time.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: issue and challenges. Computer **39**(10), 36–43 (2006)
3. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. **31**(6), 23 (2009)

4. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. J. Comput. Secur. **13**(3), 347–390 (2005)
5. Bodei, C., Degano, P., Galletta, L., Salvatori, F.: Linguistic mechanisms for context-aware security. In: Ciobanu, G., Méry, D. (eds.) ICTAC 2014. LNCS, vol. 8687, pp. 61–79. Springer, Heidelberg (2014)
6. Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for the Pi-calculus with applications to security. Inf. Comput. **168**(1), 68–92 (2001)
7. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)
8. Degano, P., Ferrari, G.L., Galletta, L.: A two-component language for COP. In: Proceeding 6th International Workshop on Context-Oriented Programming. ACM Digital Library(2014). doi:10.1145/2637066.2637072
9. Degano, P., Ferrari, G.-L., Galletta, L.: A two-phase static analysis for reliable adaptation. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 347–362. Springer, Heidelberg (2014)
10. Degano, P., Levi, F., Bodei, C.: Safe ambients: control flow analysis and security. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 199–214. Springer, Heidelberg (2000)
11. Galletta, L.: Adaptivity: linguistic mechanisms and static analysis techniques. Ph.D. thesis, University of Pisa (2014). http://www.di.unipi.it/galletta/phdThesis.pdf
12. Gasser, K.L.S., Nielson, F., Nielson, H.R.: Systematic realisation of control flow analyses for CML. In: Proceeding of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP 1997), pp. 38–51 (1997)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
14. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, 1st edn. 1999. corr. 2nd printing, 1999 edn. Heidelberg (2005)
15. Nielson, F., Nielson, H.R., Hansen, R.R.: Validating firewalls using flow logics. Theor. Comput. Sci. **283**(2), 381–418 (2002)
16. Riis Nielson, H., Nielson, F.: Flow logic: a multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
17. Salvaneschi, G., Ghezzi, C., Pradella, M.: Context-oriented programming: a software engineering perspective. J. Syst. Softw. **85**(8), 1801–1817 (2012)