A Two-Component Language for Adaptation: Design, Semantics, and Program Analysis

A Two-Component Language for Adaptation: Design, Semantics and Program Analysis

Pierpaolo Degano, Gian-Luigi Ferrari and Letterio Galletta

Abstract—Adaptive systems are designed to modify their behaviour in response to changes of their operational environment. We propose a two-component language for adaptive programming, within the Context-Oriented Programming paradigm. It has a declarative constituent for programming the context and a functional one for computing. We equip our language with a dynamic formal semantics. Since wrong adaptation could severely compromise the correct behaviour of applications and violate their properties, we also introduce a two-phase verification mechanism. It is based on a type and effect system that type-checks programs and computes, as an effect, a sound approximation of their behaviour. The effect is exploited at load time to mechanically verify that programs correctly adapt themselves to all possible running environments.

Index Terms—Adaptive Software, Context Oriented Programming, Formal Methods, Datalog, Functional Programming, Semantics, Type Systems, Verification

1 Introduction

Modern software systems are designed to operate *always* and *everywhere*. Their structure is therefore subject to continuous changes that are unpredictable at design time. A suitable management of these changes should maintain the correct behaviour of applications and their non-functional properties, e.g. quality of service. Effective mechanisms are thus required to *adapt* software to changes of the operational environment, namely the *context* in which the application runs.

The notion of *context* is fundamental for adaptive software. It includes any kind of computationally accessible information coming both from outside (e.g., available devices, code libraries etc. offered by the environment), and from inside the application boundaries (e.g., its capabilities, user profiles, etc.). The contents of the context depend on the architectural model to which software adheres. For example, in the Cloud Computing paradigm [77], the context contains at least the description of the computational resources offered by the cloud provider, and also a measure of the available portion of each resource, and the way these are partitioned for multi-tenancy. In Service-Oriented Computing (SOC) [73], software components called services are equipped with suitable interfaces describing the offered computational facilities. Standard communication protocols (e.g. SOAP over HTTP) take care of the interactions between the parties. The context is then determined by the various repositories where services are published, and by the end-points where services are actually deployed and made available, as well as by other information about service level agreement. In the Internet of Things [7], the context (partially) represents the active space that hosts and is made of the interacting digital devices, or even physical ones, like sensors. Each device has its own context.

The development of adaptive systems requires a variety of design and programming abilities, and it often involves actions directed at the different collections of hardware and software resources supplied by the environment. As pointed out by Baresi et al. [10] and by Kamina et al. [59], traditional software engineering methodologies adopt a *static* model of software development, where the boundaries between specification and development are rigidly fixed; the interactions with the operational environment are assumed *a priori*; and software reconfiguration usually occurs *off-line*. This approach has become inadequate, since applications now run in partially known and ever changing contexts. Therefore, we agree with Baresi et al. [10] and with Kamina et al. [59] that an effective development of adaptive software requires a strong synergy between the methodologies, the development and analysis tools, primarily programming languages. Indeed, the design, the development and the verification of systems can be made more effective if these are programmed with a special-purpose language, with high-level constructs for expressing adaptation patterns. The provided linguistic abstractions impose a good practice to programmers, with a positive effect on correctness and modularity, mainly because low level details are masked. Correctness is made harder to address with standard techniques, because adaptive applications are prone to a new class of run time errors: a failure may occur because the running application has not been designed for the actual hosting context, e.g. when a lacking facility is instead assumed to be available.

In this paper, we propose linguistic mechanisms for adaptivity, and we discuss the rationale for them; their static and dynamic semantics; their formal properties and a way to mechanically verify some of them, in particular absence of adaptation errors; and a running prototypical just-in-time compiler. We follow the approach of Context

The authors are with the Dipartimento di Informatica, Università di Pisa, 56126 Pisa, Italia
E-mail: {degano,giangi,galletta}@di.unipi.it
Work partially supported by the MIUR Prin Project Security Horizons *and by the Università di Pisa PRA project* Through the Fog.
Corresponding author: Letterio Galletta.
Manuscript .

Oriented Programming (COP), proposed by Costanza [34] as the main paradigm to develop adaptive software in a modular fashion.

The kernel of our proposal is the two-component language ML_{CoDa}. The first constituent is for the context, and it is the logical language Datalog with negation [29]. The second component is a core of ML, extended with powerful primitives for context management and for adaptation. The name of our language ML_{CoDa} comes from ML and *CO*ntext in *DA*talog.

Since adaptive applications may misbehave because at design time an unknown environment was not considered, we equip ML_{CoDa} with a static analysis to ensure that this kind of run time errors never occur, e.g. one arising because the actual hosting environment lacks a required capability.

The analysis is performed in two phases: a type and effect system (at compile time) and a control flow analysis (at load time). When type-checking a program we also compute an over-approximation of the capabilities that the application will need at run time. When entering a new context, *before* running the program, this abstraction is exploited to check that the actual context, and those resulting from its evolution, support the capabilities required by the application. As discussed later on, this last analysis can only be done at load time, because at compile time the possible hosting contexts are still unknown.

Structure of the paper

The next section surveys the main approaches to adaptivity proposed in the literature, with particular attention to the Context Oriented Programming paradigm. The main design choices of ML_{CoDa} are illustrated in Section 3. We justify here why we keep the context, a Datalog knowledge base, apart from the applications, expressed as ML programs, enriched with two main primitives for adaptation. The same section also presents the programming and the execution models of ML_{CoDa}, where a predefined API is provided by the virtual machine of the language, and anticipates the need of a two-phase static analysis for guaranteeing certain properties of adaptive software. The main features of the language are then intuitively introduced in Section 4, through a running example. The formal semantics for the new primitives of ML_{CoDa} is in Section 5; we assume here the standard Datalog semantics. Our type and effect system is in Section 6, along with the statements of its correctness (Theorem 6.4), entailed by the following properties: preservation (Theorem 6.1), progress (Theorem 6.2), and correctness of effects, i.e. that they over-approximate the run time behaviour (Proposition 6.3). However, the property of progress holds if adaptation is always possible: Section 7 introduces a Control Flow Analysis (and its properties, i.e. Theorems 7.1 and 7.2) that checks effects and guarantees that adaptation never fails. We also present a worklist algorithm that implements this load time static analysis. A prototypical implementation of ML_{CoDa} is briefly presented in Section 8. Section 9 concludes, briefly compares our work with the literature, discusses the main limitations of our approach and how to address them in future work. The Appendix contains all the rules for the semantics of ML_{CoDa}; the complete logical presentation of our type and effect system and the full proofs of the properties of our static analyses. [1]

2 Adaptivity in Programming Languages: A Brief Survey

In 2001 IBM released a manifesto [52], that promoted an overall rethinking of computing systems. Future systems are demanded to decrease the human involvement and to increase self-management (*autonomic computing* [60], [51]). In this vision a system continuously monitors its own execution and adapts to its working environment.

The problem of adaptability is one of the hardest research challenges in designing and building autonomic systems. Since the pioneering work on dynamic software architectures by Magee and Kramer [65], and the more recent by Kephart and Chess [60] that propose the standard architectural style of autonomic elements, a great deal of work witnesses the importance of this topic in different research areas of software engineering. Among these we only mention requirements engineering [21], software architecture [45], component-based development [74], formal foundations [22] and programming languages [80], [38].

A natural way of implementing adaptivity and handling context-awareness with standard programming languages consists of modelling the context through a special data structure, that can answer a fixed number of queries, tested through `if` statements. This approach has however at least two drawbacks. First, the programmer himself is responsible for implementing the data structure and the queries representing the context. In addition, the adaptation logic cross-cuts the application logics because the former is often orthogonal to standard modularisation mechanisms. So implementing contexts and achieving a good separation of cross-cutting concerns are not adequately supported by standard programming languages. Cross-cutting is usually solved by exploiting special design patterns that encapsulate the context-depended behaviour into separate objects. The context then drives their instantiation. However, these approaches do not foster the development of automatic verification tools.

A first linguistic approach to adaptivity is dynamic Aspect-Oriented Programming [67], [16], [88], for which many implementations are available [62], [85], as well as foundational studies [92], [93], [63]. Essentially, this proposal separates and organises orthogonal code in standalone modules, called *aspects*, by decoupling them from the application logic. Dynamic Aspect-Oriented Programming at run time allows one to remove from and to add to programs also aspects previously unknown, so adapting the system behaviour. Although Aspect-Oriented Programming adequately supports dynamic adjustments of programs, it does not explicitly represent the working environment hosting the application. Consequently, programmers themselves need to implement the context and the mechanisms to intercept and react to its changes. For this reason, we rather followed the line briefly discussed next.

Context Oriented Programming (COP) was proposed as a viable paradigm to developing context-aware software by

1. A preliminary version of Sections 3, 4 and 5 appeared in [41], and of Section 6 in [40].

Costanza and Hirschfeld [35] and by Hirschfeld et al. [49]. It advocates languages with suitable constructs for adaptation to express context-depended behaviour in a modular manner. In this way adaptivity is built-in, and mechanical verification becomes feasible to assure that program behaviour keeps its correctness after the adaptation steps. Furthermore, the code of programs can be optimised by the compiler or by the virtual machine.

There are two fundamental constructs in the proposed COP languages: *behavioural variations* and *layers*. A behavioural variation is a chunk of behaviour that can be dynamically activated through a *dispatching* mechanism, depending on information picked up from the context, so implementing adaptation. Since multiple behavioural variations can be active at the same time, the result of their combination determines the actual program behaviour.

A context is implicitly characterised by a set of layers, that can be activated and deactivated at run time. Each layer is a module that includes context dependent behaviours: activating/deactivating a layer corresponds to activating/deactivating the corresponding behavioural variation. One can simply and intuitively view a layer as an elementary property, actually as a proposition about the current context.

Since the pioneering work by Costanza [34], a large number of COP proposals emerged, all satisfying the requirements put by Hirschfeld et al. [49]. Roughly, these characterise the features that any COP language must possess in order to:

- specify behavioural variations
- group them in well defined and isolated layers
- dynamically activate/deactivate behavioural variations, depending on the current context
- explicitly and dynamically control the scope of behavioural variations

Additionally, Hirschfeld et al. [49] advocate layers to be first-class objects in the language, allowing them to be bound to variables, passed as argument to and returned by functions. This feature is especially required to allow different parts of a system to communicate and to adapt at run time.

We briefly survey below the most significant design choices concerning *activation mechanisms* and *behavioural variation modularisations* proposed in the literature.

Usually, the layer activation strategies can be divided in *global* or *local*. In the first case, there exists a unique shared context and a behavioural variation activation affects the control-flow of all threads. In the second case, there exist distinct local contexts, e.g. for groups of threads or objects of the application, and the activation only affects the behaviour of the entities that depend on the modified context.

Regarding the *extent of variation activation* we can identify two approaches: *dynamically scoped activation*, the most common, and *indefinite activation*. In the first approach activation is performed by a statement of the form `with(layer){statements;}`, which activates `layer` while evaluating `{statements;}`. The activation affects also the possible nested calls in a stack-wise discipline. A newly activated layer is pushed on the stack recording the active layers, and it is popped as soon as its scope expires. In this way the previous configuration is retrieved. This strategy allows a programmer to sharply specify which parts of the program can adapt their behaviour. Following the indefinite activation strategy, the programmer himself has instead to specify when a layer is activated or deactivated, through special statements.

The *behavioural variation modularisation* depends on the strategy for *layer declaration*, either *class-in-layer* or *layer-in-class*. The first strategy denotes declarations where the layer lexically surrounds the modules for which it defines the behavioural variations in hand. Actually, layers are encapsulated in ad hoc, isolated modules. The second declaration strategy, instead, supports the declaration of a layer within the lexical scope of the module it augments, so that the definition of a module is completely specified.

So far most research efforts in the field of Context-Oriented Programming have been directed toward the design and the implementation of concrete languages. The survey by Salvaneschi et al. [81] discusses in detail the design of languages, and that by Appeltauer et al. [5] analyses some implementations. Below, we focus on some papers that formally study constructs for adaptivity within COP, in particular those supporting verification.

ContextFJ [32] extends Featherweight Java [54] with layers, scoped layers activation, and deactivation. Clarke and Sergey have not considered constructs for expressing inheritance and have adopted a class-in-layer strategy [49] to express behavioural variations. Since layers may introduce methods not appearing in classes, they have also defined a static type system ensuring that there exists a binding for each dispatched method call.

A different model, based on Featherweight Java, is introduced by Hirschfeld et al. [50]. It includes inheritance and still exploits a class-in-layer strategy. Also in this case, a type system has been specified to statically prevent erroneous invocations at run time. The type system of Hirschfeld et al. [50] is much more restrictive than that of Clarke and Sergey [32], because it prohibits layers from introducing new methods that do not exist in the class. This means that every method defined in a layer has to override a method with the same name in the class. This restriction is addressed by Igarashi et al. [53], who introduce a type system handling dynamic layer composition. Furthermore Hirschfeld et al. [50] has no construct for deactivating a layer, and Kamina et al. [56] addresses this problem through the so-called *on-demand activation*, the semantics of which is given by extending ContextFJ. The proposed mechanisms activate all the layers required by the current context, so avoiding errors due to non-activated, yet required layers. The type system of ContextFJ is extended by Inoue et al. [55] to include the main features of JCop [6], a Java-like language. In particular, it handles first-class layers, their inheritance and subtyping between layer types. LamFJ [4] is a variant of ContextFJ that offers most of the different layer activation mechanisms proposed in the literature and that combines their effects uniformly. Its new features are called *context change events* and *event firing expressions*, through which one can express the most common COP activation mechanisms. The first represents activation and deactivation of layers. The second are special expressions describing which context change events are to be fired. Featherweight EventCJ [3] is another calculus inspired by Featherweight Java, that has been introduced to formalise event-based layer activation.

This mechanism allows activating layers reacting to events triggered by the environment. The notions of *atomic layer*, *composite layer* and *implicit layer activation* are added to Featherweight EventCJ by Kamina et al. [57]. An atomic layer has the usual form, and is explicitly activated by the programmer. A composite layer contains a propositional formula in which ground terms represent other layers (true when active). The programmer has no control on the activation of a composite layer, that instead is implicitly activated when the formula it contains holds.

Contextλ [31] extends the λ-calculus with layer definition, activation/deactivation and a dispatching mechanism. However, the expressions of the calculus do not include higher order behavioural variations. Contextλ is designed to study the issues deriving from the combination of closures and the special `proceed` construct, a sort of `super` invocation in object oriented languages [49]. The problem arises when a `proceed` appears within a closure that escapes the context where it was defined. This opens interesting semantic issues because escaping from a context may cause the application to live in a context where the required layers could not be active any longer. In Clarke et al. [31] several ways to deal with this semantically relevant problem have been proposed, yet to the best of our knowledge, the question is still open.

A predecessor of ML_{CoDa} is ContextML [42], that extends ML with layers, layered expressions, and scoped activation mechanisms for layers (`with` and `without`). It is endowed with a type and effect system, and effects are model-checked to enforce communication compliance and security policies.

Cardozo et al. [27] propose a run time verification of adaptive programs based on dynamic symbolic execution. Their idea consists of a verification step before activating/deactivating a layer, in order to check whether adaptation is possible. A different approach is discussed by Cardozo et al. [28], that represents the structure of contexts as a(n enriched) Petri net. Relying on the dynamics of nets and on existing verification tools, the authors show how to analyse properties of the activation state of contexts.

We now focus on how the context is modelled in approaches outside of the COP paradigm. In some cases, the context is a computational entity with its own data model, and rigidly separated from the programming language. Indeed, some proposals introduce a *context manager* that essentially acts as an interface between the application and the context hosting it. Answering queries of applications is a main task of a context manager, and thus an application has to take care of data communication and serialisation. In Chen et al. [30], Wang et al. [94] and Gu et al. [47], the context is a collection of ontologies, specified in suitable description logics. We avoid the problem of different data representations in the context and in the application [66] (impedance mismatch), because our two-component language has a *single* data model, implemented by the virtual machine. Also other proposals, e.g., those by van Wissen et al. [90] and by David and Ledoux [37] aim at mitigating the impedance mismatch by representing the context through objects. However, in these last approaches it may be hard to make complex queries, involving convoluted deductions. The Datalog machinery is an asset of our proposal.

There is also a great deal of work on adaptivity in the event-based and agent-based approaches. Here we do not discuss either of them, and we only refer the reader to the paper by Bordini et al. [18] and to Bainomugisha's PhD thesis [8].

3 The Design of ML_{CoDa}

We propose ML_{CoDa} a core functional programming language, equipped with linguistic primitives for context-awareness, coupled with a logic language, for context definition and management.

Our first goal is the definition of high-level primitives for describing complex working environments, with components taking values on rich domains. Such primitives could provide non experts with a more intuitive and succinct notation, fostering early prototyping. The mechanism selecting the suitable behaviour can be tuned on a rich variety of values, and not only on the single basis of the relevant layer being active, as it is often the case with many implementations.

Secondly, we wish to have high-level linguistic constructs to have powerful means to drive adaptation. Even though behavioural variations are a primary notion in COP, they are often expressed as (partial) definition of procedures or classes or methods. In some proposals they are first-class objects, e.g. in ContextL [35], but in many others they are not, e.g. in Java-based languages. We propose higher-order behavioural variations, that in our view may help programming dynamic adaptation patterns, as well as reusable and modular code, because they can be bound to variables, passed to functions, and manipulated with specific operators. For example, an autonomic element [60] can take a behavioural variation from the context, and compose it with existing ones, so offering a further way of adapting its behaviour (see e.g. the function `addDefault` in the next section).

Our third goal is to propose verification mechanisms for COP languages firmly built on foundational bases. Therefore, another main contribution of this work is providing ML_{CoDa} with a two-phase verification mechanism, proved correct against the expected formal dynamic semantics. Verified programs are prevented from running when they are not capable to adapt to the current context and to its possible modifications. Of course more flexible ways of handling errors can be defined on top of our analysis, that we do not discuss in this paper.

3.1 Representing the context

Our first concern in the design of our COP language is the context, a formal description of the environment where applications run.

Intuitively, the context is a heterogeneous collection of data coming from different sources and having different representations. Some of these data are application independent, like those about the hardware capabilities, e.g. the screen resolution, and about the physical environment, e.g. the level of noise in the location of the user; other data are application-dependent like user's preferences, e.g. accessibility options. Furthermore, the separation between data coming from inside the application and those coming from

outside is a well-established practice. For example, Salehie et al. [79] advocate distinguishing between what they call the *context* and the *self*. The first identifies the properties of the operational environment, the second describes the properties of the application itself.

Following this approach, we split our context in two coarse parts: the *system* and the *application context*. Both parts are represented and manipulated in a uniform way. The system context pertains to the environment running the application, for example to the virtual machine of the language. It is accessible through a predefined API whose actual data are only available at run time (see also the execution model below). The application context stores specific knowledge of the application, and its contents are given by programmers. Of course, the application context can use information from the system context. Therefore, the actual context of the application at run time results from combining the system and the application contexts. Note that we assume programmers to only interact with the system running the application via the API, that makes code and data available to them.

It is worth noting that the context influences the shape and the features of the program input (e.g. from where it is taken) and how it is processed, but the context is *not* part of the input itself. Of course, the context affects also the output of the application.

A programmer needs tools to access and manipulate all kind of contextual data in a easy and uniform way. Indeed, programming the context requires tools and skills different from those needed for building applications [72], [64]. This methodological issue, as well as separation of concerns motivate us to define ML_{CoDa} as a two-component language: a declarative constituent for programming the context and a functional one for computing.

The declarative approach allows programmers to express *what* information the context has to include, leaving to the virtual machine *how* this information is actually collected and managed. For us, a context is a knowledge base and we implement it as a Datalog program, following a well-studied approach (see e.g. Orsi et al. [72], Loke [64], and Alvaro et al. in a distributed setting [1]). In other words, a context in ML_{CoDa} is a set of facts that predicate over a possibly rich data domain, and a set of logical rules that permit to deduce further implicit properties of the context itself. With this representation, adaptive programs can query the context by simply verifying whether a given property holds in it, i.e. by checking a Datalog goal. Note that deduction in Datalog is fully decidable in polynomial time [29]. During the needed deductions relevant information is also retrieved.

Our notion of context conforms to the *COSE* methodology proposed by Kamina et al. [59]. We generalise their hypotheses over the context to be a set of boolean variables. Indeed, we represent it as a set of Datalog predicates, the terms of which are interpreted as contextual data. Accordingly, we also provide mechanisms to create abstractions for managing the context: Datalog rules allow us to easily define high level properties by aggregating existing ones and to extend them by simply adding new definition cases. Furthermore, our approach can be integrated in the *CODA* methodology by Desmet et al. [44] proposed for modelling context-aware software requirements. In particular, *CODA*

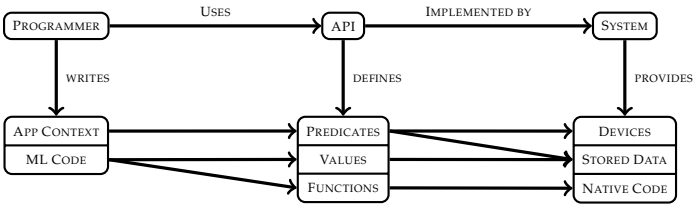

Figure 1. The programming model of ML_{CoDa}

diagrams can be easily mapped into Datalog rules, instead of decision tables.

3.2 Specifying adaptation

As for programming adaptations, we propose two mechanisms. The first is *context-dependent binding*, a mechanism through which a programmer declares variables whose values depend on the context. For that, we introduce the `dlet` construct that is syntactically similar to the standard `let`, but has additional Datalog goals therein. The variable declared (called *parameter* hereafter) may denote different objects, with different behaviour depending on the different properties of the current context, which is a major aspect of adaptivity.

The second mechanism is based on *behavioural variations*, the fundamental concept of the COP paradigm. As said above, a behavioural variation is a chunk of behaviour that can be activated depending on information picked up from the context, to dynamically adapt the running application. Roughly this new construct is a list of pairs `goal.expression`, similar to pattern-matching, that alters the flow of the application depending on the context. When a behavioural variation is executed, parts of the deployed code are suitably selected to meet the new requirements. Behavioural variations have parameters, and they are *first-class values*.

We propose mechanisms for adaptation as flexible as possible but not *too flexible*. Our choice is supported by the following methodological principles: (i) "although developers cannot foresee all changes, they must define the boundaries the system can evolve within, and the criteria it will consider in deciding how to evolve. Without defining these, self-managing systems would soon go out of control" [10]; (ii) "predictable control of changes of context-dependent behaviour is also important" [59]. Accordingly, our constructs require programmers to identify how the context may change, and to set up the alternatives to be used at run time. Nevertheless, our behavioural variations can be manipulated and composed at run time to build new ones, but always in a type-safe way and within the boundaries defined at design time.

3.3 Programming and execution models

We now discuss our programming model, displayed in Figure 1 (the labels Uses and Implemented by extend downwards to all unlabelled arrows). We assume that the virtual machine of the language provides its users with a a predefined API, that offers a collection of (the signatures of) system variables, values, functions and predicates, e.g. the

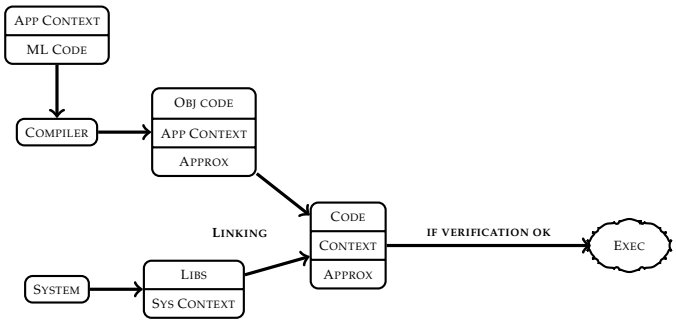

Figure 2. Execution model of ML_{CoDa}

functions and predicates for interacting with an accelerometer of the device running the application. Recall that the context is split in two parts: the system and the application contexts. The first is made available to the application by the API, and it is thus implemented by the virtual machine, while the the programmer fills in the application context. Obviously, programs acquire information about the system context through system predicates, but we stress that the actual values holding therein are only available at run time. In this paper we are not interested in the actual form of the API and so we do not further detail it.

In our execution model, shown in Figure 2, the compiler produces a triple (C, e, H), where C is the application context, e is the program object code and H is an approximation of e, used to verify properties about the program. Here, we are specifically interested in checking whether an application will adapt to all the contexts that may arise at run time. Therefore, we will record in H all the actions performed by e on the context that will host it. More precisely, H will contain (a superset of) all the queries and of all the updates possibly carried out by e. Given such a triple, at load time the virtual machine performs a *linking* and a *verification* phase. The linking phase resolves system variables and links the application and the system contexts, thus obtaining the initial context that, of course, is checked for consistency in the logical terminology. Note that linking itself makes a first step of adaptation, because it enables the application to use the capabilities of the hosting system, be they resources, data or code. In the spirit of Proof-Carrying code [69] and of the Java Bytecode Verifier [78], the verification phase exploits the approximation H to check that the program e will adapt to *all* the contexts that may occur at run time. If both phases succeed, program evaluation begins, otherwise in this basic model it is aborted. Our verification machinery borrows from those by Skalka et al. [83] and by Bartoletti et al. [14], in that we construct an abstraction of program behaviour through a type and effect system and then we use the abstraction to check properties of the code. Indeed, those approaches use a machinery based on a linear time model-checking, whereas ours differs in two aspects. First, our verification cannot be completed at compile time because contexts are only known at load time. Secondly, we do not model-check abstractions, but we use them to construct a graph which describes how the initial context changes over time.

4 A Guided Tour of ML_{CoDa} Features

In this section we illustrate and discuss the main features of ML_{CoDa} through a running example, with some aspects typical of the Internet of Things. Consider a smartphone application implementing a multimedia guide to museums. First a user registers at a desk and gets credentials; he then uses them and connects to the museum Intranet to download the guide application for his smartphone. The Intranet provides communication facilities and hosts a website, with a page for each exhibit with relevant information, e.g. videos documenting a recent restoration. First, we briefly overview the functionalities of the application and then we discuss some snippets of code. For readability, in the examples below we use a sugared syntax for ML_{CoDa}(but the specification of the context is in standard Datalog [72]). The functional component extends a core of ML and inherits most of its constructs and behaviour. The full definition of the syntax of ML_{CoDa} is in Figure 3, while that of the semantics of its functional component is in Appendix A (again, the semantics of Datalog is standard).

4.1 A Multimedia Guide for Museums

We assume the museum has a wireless infrastructure exploiting different technologies, like WiFi, Bluetooth, Irda or RFID. Each exhibit is equipped with a wireless adapter and a QR code that are only used to supply the guide with the URL of the exhibit. The way this URL is retrieved depends on the visitor's smartphone capabilities. For example, the URL is directly downloaded by Bluetooth, if such a device is available; otherwise if the smartphone has a camera and a QR decoder, the guide can decode it and retrieve the URL.

There are tickets with different prices, depending on the profile provided by the visitor during registration; e.g. if he is a European citizen over 65, the reduced fare applies. After buying the ticket, the visitor can set some preferences, including accessibility options. For example, a user can choose to only have textual information or to have all texts read by a speech synthesiser. The guide then supplies the user with the tour, taking into account his preferences and information about the physical environment acquired by the sensors of his smartphone.

4.2 The Context

The most important design choice in the development of a context-aware application is deciding what the context is. As argued in Section 3 the context is split in a part for the system and another for the application.

As for the system context, Datalog predicates and facts represent properties and capabilities of devices. For example, the *system predicate* `headphones` tests whether headphones are plugged in the smartphone. If this is the case, the fact

```
headphone(plugged)
```

will hold in the system context. To check a particular feature of a device, a programmer simply queries the context by the standard Datalog machinery, and needs not to care about any low-level native code to interact with hardware.

Note that predicates not only represent devices (and their implementations), but also provide the programmer

with an interface for interacting with the software services offered by the system.

Besides system predicates that are independent of the specific application, the programmer specifies the *application* context.

In our example, the multimedia guide stores and accesses data about the user's profile, preferences and accessibility options. For instance, if the user prefers to display all information as text or he declares to be blind, either of the following facts will hold in the application context

```
user_prefer(text_only)
user_acc_opt(blind)
```

So far, our predicates are simply facts. However, the definition of aggregated data, their retrieval and tests on them may require some deductions, because the contextual information comes from different sources. To this aim, Datalog clauses and its deduction machinery are clearly an asset of our proposal.

Predicates `only_text` and `only_speech` concern which media provide the user with the information required:

```
only_text() ← user_prefer(text_mode).
only_text() ← user_acc_opt(deaf).
only_text() ← level_noise(x),
              x > noise_threshold,
              ¬ headphones(plugged).

only_speech() ← speech_synthesizer(on),
                user_prefer(speech_mode).
only_speech() ← user_acc_opt(blind).
```

The predicate `only_text` holds whenever the user only wants textual information or if he declared hearing impairments. Note that in the third clause, we use the system predicate `level_noise` that encapsulates a routine computing the level of noise in the room where the smartphone is. We assume that this routine, as well as the `headphone` predicate are supplied by the API. Note that measuring the level of noise may require complex operations, like interacting with the microphone and executing a numerical algorithm. We check if the value returned by `level_noise` exceeds the given `noise_threshold`. Resolving a goal containing the predicate `only_text` requires then a non trivial deduction in Datalog, consisting of several steps, each for the above mentioned operations.

We list below more clauses, part of the application context of our example.

```
video(hd) ← screen_quality(hd),
            supported_codec(H.264),
            ¬ battery_level(low).
...
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).

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

The HD video format, defined in the first clause, requires the smartphone to support high definition, to run a codec, and to have enough battery power. The others predicates describe capabilities of the smartphone for retrieving URLs. The first predicate holds when the smartphone can decode QR codes; the second one if direct communication is possible. Most of the predicates used above are given by the system through its API, e.g. `device`.

4.3 The application behaviour

We now discuss the two main constructs of ML_{CoDa}, *context-dependent binding* and *behavioural variations*. They are used to program how applications adapt to changes in the context.

4.3.1 Context-dependent binding

Assume that the GUI of the multimedia guide gets an additional text label to display information about exhibits, unless the user prefers no textual information. This is implemented as follows

```
dlet txt_label = getLabel ()
      when ← ¬only_speech() in
(* other code *)
```

If the parameter `txt_label` is referred to in a context where `only_speech()` does not hold, the function `getLabel` is called and binds the returned value to the current occurrence of `txt_label`. Note that `getLabel` is *not* called when `txt_label` is declared, but when it *is used*, similarly to a call-by-name evaluation. To better illustrate this mechanism, consider the following snippet of code in ML, that adds components to the main window of the application

```
fun setMainWindow window =
    (* create and set menu *)
    addComp window txt_label;
    addComp window vcanvas;
    (* add other components *)
```

If the goal `¬only_speech()` holds in the context where `setMainWindow` is running, then `getLabel()` is evaluated and the returned value bound to `txt_label`. Otherwise, an adaptation error occurs, because `txt_label` gets no value: our load time analysis will detect this kind of run time error.

We now exemplify the multiple declaration of a parameter representing the canvas where the guide will display videos. Different kinds of canvas can be selected, depending on the quality (e.g. high or low) of videos to reproduce and on the smartphone capabilities. Below, the parameter `vcanvas` gets a multiple declaration, and the appropriate one will be selected when the above `addComp window vcanvas` is run (the second `dlet` is nested in the first one)

```
dlet vcanvas = getHDCanvas ()
  when ← video(hd), ¬only_text() in
  dlet vcanvas = getLowQualityCanvas ()
       when ← video(low), ¬only_text() in
          (* other code *)
```

For simplicity, in the snippet above, we used twice the predicate `¬only_text()`. A more economic implementation can be obtained by resorting to higher-order or nested behavioural variations, illustrated in the next paragraph.

We now discuss the main difference between our context-dependent binding and standard dynamic binding. Consider the following snippet of code

```
(* definition of setMainWindow & vcanvas *)

fun createMainWindow () =
  (* create window *)
  setMainWindow window
  (* other code *)
```

where the function `setMainWindow` is defined before the declarations of the parameter `vcanvas` and then it is applied in the body of the function `createMainWindow`. Assume your smartphone can reproduce high-definition videos and that the current context grants the goal $\texttt{video(hd)}, \neg\, \texttt{only\_text}()$, but not $\texttt{video(low)}, \neg\, \texttt{only\_text}()$. Now, call `createMainWindow`, which in turn calls `setMainWindow`: the parameter `vcanvas` is bound to the value returned by `getHDCanvas`. With dynamic binding, `vcanvas` would instead be bound to the value returned by `getLowQualityCanvas`.

4.3.2 Behavioural variations

The other main feature to express context-dependency is that of *behavioural variation*, which alters the application flow depending on the context. Roughly this new construct is a list of guarded expressions `goal.expression` enclosed by curly brackets (the dot separates the guard from the expression). It is similar to pattern-matching, but with goals instead of patterns. Additionally, behavioural variations may have parameters to which they are applied, similarly to functions.

As an example, consider the following behavioural variation `url` (according to the syntax of Figure 3, the function will have a dummy argument "_" omitted here)

```
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
```

Intuitively, the list of pairs `goal.expression` is visited in textual order, and the goals evaluated at run time through a query on the context. The first expression whose goal holds is then selected (other subsequent goals may hold, but the expressions they guard are ignored; we shall come back on the evaluation order in the conclusion). If no goal holds, then a run time error occurs: the application cannot adapt, because the current context does not enjoy a desired property, e.g. it lacks a required capability. Following the usual COP terminology, we call this mechanism *dispatching*.

Depending on the smartphone capabilities, the behavioural variation above retrieves the URL of an exhibit page. If communication with the exhibit adapter is direct, the application reads the URL through the channel returned by `getChannel`; otherwise, the smartphone takes a picture of the QR code and decodes it. Note that in this second case the variables `decoder` and `cam` will be assigned the handles of the decoder and the one of the camera deduced by the Datalog machinery. These handles are used by the functions `take_picture` and `decode_qr` to interact with the actual smartphone resources.

The behavioural variation (bound to) `url` is applied before invoking the function `getRemoteData` (for readability, here we use a simplified syntax for behavioural variation application represented by #; for details see Section 5).

As a further example of behavioural variation, consider the function `getRemoteData` that connects to the website and downloads the page of the exhibit:

```
fun getRemoteData url =
(* other code *)
  ← ¬only_speech().
   (* other code *)
     let img = {
       ← orientation(landscape),
         sscreen(large),
         supported_media(png).
         getImg(url + "/" +
                iname + "-large.png"),
       ← orientation(portrait),
         sscren(small),
         supported_media(svg).
         getImg(url + "/" +
                iname + "-small.svg")
          (* other code *)
       }
     in
       (* other code *)
```

The actual downloaded data depend on the preference of the user and on the display capability of the smartphone.

It is worth noting that behavioural variations are values, so we can write code that manipulates them. This allows programmers to implement adaptation patterns they think more appropriate for their applications and write more modular and extensible code.

To manipulate behavioural variations we equip ML_{CoDa} with the binary operator $\cup$ that extends the cases over which a behavioural variation is defined by concatenating the lists of pairs `goal.expression` of another behavioural variation. As an example of behavioural variation concatenation, let `True` be the goal always true, and consider the function

```
fun addDefault bv dv =
    bv ∪ { True. dv }
```

It takes as arguments a behavioural variation `bv` (with no parameter) and a value `dv`, and extends `bv` by adding a default case which is always selected when no other case apply. Note in passing that this function implements a sort of "extensible" programs pattern. In particular, the above `addDefault` allows ML_{CoDa} programmers to easily implement the standard notion of *basic behaviour* in COP languages.

4.3.3 Context updates

Besides the features that describe and query the context, and those that adapt program behaviour, ML_{CoDa} is also equipped with constructs that update the context by adding facts, `tell`, and removing them, `retract`.

For example, the function below inserts in the context the fact returned by `getCheckedOption` with the argument `accRadioButton`:

```
fun setAccessibilityOpt () =
   tell ( getCheckedOption accRadioButton )
```

As it will be clearer later on, the type of `getCheckedOption` will record that it returns either fact `user_acc_opt(deaf)` or `user_acc_opt(blind)`. Thereby, it is possible to predict how the context will be affected by the function `setAccessibilityOpt` (same if `retract` replaces `tell`).

Note that `tell` and `retract` belong to the functional component of ML_{CoDa}, and they can be used neither in the context description nor in the goals within `dlet` and within behavioural variations. This implies that querying the context, i.e. making deductions, has no side-effects. Furthermore, with these constructs the programmer can insert or remove facts of the application context, only. Instead, the system context can only be updated through the API provided, so it is possible to prevent the programmer from driving the system in an invalid state, e.g. by inhibiting access to a hardware device if the battery level is too low.

4.4 Adaptation failures

An application can fail to adapt to a context, both because the programmer assumed at design time the presence of functionalities that the current system context lacks, or because of programming errors. This is a new class of run time errors that we call adaptation failures.

In ML_{CoDa} this may happen when some crucial facts turn out to be unexpectedly false in the current context, e.g. because they have been removed. In this case a parameter in a context-dependent binding cannot be resolved or a behavioural variation gets stuck, i.e. the dispatching mechanism finds no viable alternative to run and fails. The common reason is that the goals occurring in these constructs do not hold.

Back to our example, consider the function `setMainWindow`. While evaluating it, `addComp` is called twice, with the parameters `txt_label` (in the first call) and `vcanvas` (in the second one). Assume that the context satisfies neither goals in the declarations of `vcanvas`, so it cannot be resolved and a run time error occurs. The dispatching mechanism fails, e.g., while evaluating `getExhibitData` on a smartphone without wireless technology and QR decoder. Of course, no context will ever satisfy the goals of the behavioural variation `url`. So when `url` is applied, no case can be selected.

An overly conservative approach to avoid these run time errors could be disallowing code that fails to run under any possible context. Our proposal tries to detect whether an application will be able to adapt to its execution context right before running in it. To do that we equip ML_{CoDa} with a two-phase static analysis: one at compile time and the other one at load time.

At *compile time* we associate a type and an effect with an expression e. The type is (almost) standard, and the effect is an over-approximation of the actual run time behaviour of e, and abstractly represents the changes and the queries performed on the context during its evaluation.

At *load time* we exploit the effect computed by the type system to build a graph which describes how the context may change at run time. Visiting the graph we can detect if the queries of the application at run time may fail. Here we only use graphs for this purpose, but other properties can be analysed on them, e.g. security and access control. Also, we can single out specific points in the code in which to insert calls to suitable routines for error handling.

For example, consider the function `getExhibitData`. Its computed type is $unit \xrightarrow{H_1} \tau_d$ where τ_d is the type of the value returned by the application of `getRemoteData`. The annotation H_1 is $ask\, G_1 \otimes ask\, G_2 \otimes fail$, where $G_1 = \leftarrow \texttt{direct\_comm}()$ and $G_2 = \leftarrow \texttt{use\_qrcode(decoder)}, \texttt{camera(cam)}$ are the goals of `url` (for readability, here we use a simplified syntax for type annotations; for details see Section 6). Intuitively, H_1 says that one between G_1 or G_2 must be satisfied by the context in order to successfully apply the function `getExhibitData`, otherwise a failure occurs. At load time our static analysis ensures us that at least one among G_1 and G_2 will hold in the context.

As a further example, consider the function `setAccessibilityOpt`. Its type is $unit \xrightarrow{H_2} unit$ where the annotation H_2 is

$$\texttt{tell(user\_acc\_opt(deaf))} + \texttt{tell(user\_acc\_opt(blind))}$$

means that `setAccessibilityOpt` modifies the context by adding either fact yielded by `getCheckedOption`, i.e. `user_acc_opt(deaf)` or `user_acc_opt(blind)`.

We conclude this section with a short remark on how we positively addressed the quest for a flexible and controlled adaptivity posed by Baresi et al. [10] and by Salehie and Tahvildari [79]. In our proposal, the APIs, the system and the application contexts define the adaptation boundaries at design time. At run time, behavioural variations automatically adapt the application to the actual needs of the working environment within the boundaries.

5 The Syntax and the Semantics of ML_{CoDa}

As discussed above, ML_{CoDa} consists of two components: Datalog with negation to describe the context and a core of ML extended with COP features. Its abstract syntax (in Figure 3) and a glimpse of the structural operational semantics follow (the full definition is in Appendix A).

5.1 Syntax

Context component

The syntax of Datalog is the standard one [29] and it is displayed in Figure 3 (left). As usual, we assume to have the following sets: Var (ranged over by $x, y, ...$) for variables, $Const$ (ranged over by $c, n, ...$) for constants and $Predicate$ (ranged over by $P, ...$) for predicate symbols.

As usual in Datalog [29], a term is a variable x or a constant c; an atom A is a predicate symbol p applied to a list of terms; a literal is a positive or a negative atom; a clause is composed by a head, i.e. an atom, and a body, i.e. a possibly empty sequence of literals; a fact is a clause with an empty body and a goal is a clause with empty head.

A Datalog program is a finite set of facts and clauses. In the following we assume that each Datalog program is safe [29], i.e. it satisfies the following requirements: (i) each fact is ground; (ii) each variable occurring in the head of a clause must occur in the body of the same clause; and (iii)

Syntax of Datalog component

$$x \in Var \qquad c \in Const \qquad p \in Predicate$$

$$\begin{array}{lll} T ::= & x \mid c & T \in Term \\ A ::= & p(T_1, \ldots, T_n) & A \in Atom \\ L ::= & A \mid \neg A & L \in Literal \\ R ::= & A \leftarrow B. & R \in Clause \\ B ::= & \epsilon \mid L, B & B \in ClauseBody \\ F ::= & A \leftarrow \epsilon & F \in Fact \\ G ::= & \leftarrow L_1, \ldots, L_n. & G \in Goal \end{array}$$

Syntax of functional component

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

Figure 3. Syntax of ML_{CoDa}

each variable occurring in a negative literal must also occur in a positive literal of the same clause.

Following the usual Datalog terminology, we classify predicates into extensional and intensional. The former represent concrete context objects, like resources, users preferences and data. The latter describe relationships among, and properties about context objects. For instance, the predicate `user_prefer` of Section 4 is extensional, because it describes a user preference; while `only_text` is intensional, because it expresses a property of the context by composing properties of the users and of the objects therein. As usual extensional predicates can occur only in facts. Thus in a clause $A \leftarrow B$, the head A only contains intensional predicates, and the body B may contain all kinds of predicates, if non-empty. We can easily ensure this requirement by a syntactic analysis of a program.

To deal with negation, we assume our world to be closed and we only admit stratified programs [29].

Application component

The functional part inherits most of the ML constructs, and its syntax is in Figure 3 (right).

In addition to the usual values, ours include Datalog facts F and behavioural variations. Also, we introduce the set $\hat{x} \in DynVar$ of *parameters*, i.e., variables assuming values depending on the properties of the running context (Var are standard variables). Our COP constructs include behavioural variations $(x)\{Va\}$, each consisting of a variation Va with argument x. A variation is a list of expressions e_i guarded by Datalog goals G_i of the form $G_1.e_1, \ldots, G_n.e_n$. The variable x can freely occur in the expressions e_i. At run time, 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 $\hat{x}$ to a variation Va. The $tell/retract$ constructs update the context by asserting/retracting facts. The append operator $e_1 \cup e_2$ dynamically concatenates behavioural variations, so providing a further adaptation mechanism. 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.

5.2 Semantics of ML_{CoDa}

We now endow ML_{CoDa} with a small-step operational semantics. Recall that the evaluation of a program only starts after the virtual machine has completed the linking and verification phases. After these phases the expression is *closed*, because it has no free variables, whereas parameters may be left free.

For the Datalog evaluation we adopt the top-down standard semantics for stratified programs [29]. Given a context C and a goal G, $C \vDash G\, with\, \theta$ means that there exists a substitution θ, replacing constants for variables, such that the goal G is satisfied in the context C.

The ML_{CoDa} semantics is inductively defined for expressions with no free variables, but possibly with free parameters. These will take a value in an environment ρ, i.e. a function mapping parameters to variations $DynVar \to Va$. In the following we update this environment through the standard operator

$$\rho[b/x](y) = \begin{cases} b & \text{if } y = x \\ \rho(y) & \text{otherwise} \end{cases}$$

A transition $\rho \vdash C, e \to C', e'$ says that in the environment ρ, the expression e is evaluated in the context C and reduces to e' changing the context C to C'. We assume that the initial configuration is $\rho_0 \vdash C, e_s$ where ρ_0 contains the bindings for all system parameters, and C results from the linking of the system context $\overline{C}$ and of the application context C_p, as illustrated in Figure 2.

Most of the semantics rules are inherited from ML. Figure 4 only shows those for our new constructs, and we briefly comment on them below. The complete list of rules is in the Appendix A.

The rules for $tell(e)/retract(e)$ evaluate the expression e until it reduces to a fact F (rule (TELL1)/(RETRACT1)). Then, the evaluation yields the unit value $()$ and a new context C', obtained from C by adding/removing the fact F (rule (TELL2)/(RETRACT2)). The following example shows the reduction of a $tell$ construct, where we apply the function `getAccessibilityOpt` of the previous section to unit, assuming that `getCheckedOption` returns the fact `user_acc_opt(blind)`.

$$\begin{array}{l} \rho \vdash C,\ \texttt{setAccessibilityOpt}\ () \to \\ \quad C,\ \texttt{tell}(\texttt{getCheckedOption accRadioButton}) \to^* \\ \quad C,\ \texttt{tell}(\texttt{user\_acc\_opt(blind)}) \to \\ \quad C \cup \{\texttt{user\_acc\_opt(blind)}\}, () \end{array}$$

The rules (DLET1) and (DLET2) that handle 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

$$\text{(TELL1)}\ \frac{\rho \vdash C,\ e \to C',\ e'}{\rho \vdash C,\ tell(e) \to C',\ tell(e')} \qquad \text{(TELL2)}\ \frac{}{\rho \vdash C,\ tell(F) \to C \cup \{F\},\ ()} \qquad \text{(RETRACT1)}\ \frac{\rho \vdash C,\ e \to C',\ e'}{\rho \vdash C,\ retract(e) \to C',\ retract(e')}$$

$$\text{(RETRACT2)}\ \frac{}{\rho \vdash C,\ retract(F) \to C \backslash \{F\},\ ()} \qquad \text{(DLET1)}\ \frac{\rho[G.e_1, \rho(\hat{x})/\hat{x}] \vdash C,\ e_2 \to C',\ e_2'}{\rho \vdash C,\ dlet\, \hat{x} = e_1\, when\, G\, in\, e_2 \to C',\ dlet\, \hat{x} = e_1\, when\, G\, in\, e_2'}$$

$$\text{(DLET2)}\ \frac{}{\rho \vdash C,\ dlet\, \hat{x} = e_1\, when\, G\, in\, v \to C,\ v} \qquad \text{(PAR)}\ \frac{\rho(\hat{x}) = Va \qquad dsp(C,\ Va) = (e,\ \{\overrightarrow{c}/\overrightarrow{y}\})}{\rho \vdash C,\ \hat{x} \to C,\ e\{\overrightarrow{c}/\overrightarrow{y}\}} \qquad \text{(APPEND1)}\ \frac{\rho \vdash C,\ e_1 \to C',\ e_1'}{\rho \vdash C,\ e_1 \cup e_2 \to C',\ e_1' \cup e_2}$$

$$\text{(APPEND2)}\ \frac{\rho \vdash C,\ e_2 \to C',\ e_2'}{\rho \vdash C,\ (x)\{Va_1\} \cup e_2 \to C',\ (x)\{Va_1\} \cup e_2'} \qquad \text{(APPEND3)}\ \frac{z \text{ fresh}}{\rho \vdash C,\ (x)\{Va_1\} \cup (y)\{Va_2\} \to C,\ (z)\{Va_1\{z/x\},\ Va_2\{z/y\}\}}$$

$$\text{(VAAPP1)}\ \frac{\rho \vdash C,\ e_1 \to C',\ e_1'}{\rho \vdash C,\ \#(e_1,\ e_2) \to C', \#(\ e_1',\ e_2)} \qquad \text{(VAAPP2)}\ \frac{\rho \vdash C,\ e_2 \to C',\ e_2'}{\rho \vdash C,\ \#((x)\{Va\},\ e_2) \to C', \#((x)\{Va\},\ e_2')}$$

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

Figure 4. Semantic rules of the new constructs for adaptation of $\mathsf{ML}_{\mathsf{CoDa}}$

rule (DLET1) extends the environment ρ by appending $G.e_1$ in front of the existing binding for $\hat{x}$. 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 *dlet* reduces to the value eventually computed by e_2.

The (PAR) rule looks for the variation Va bound to $\hat{x}$ in ρ. Then we introduce a mechanism to dynamically resolve parameters and behavioural variations, that we call *dispatching* mechanism following the standard COP terminology. The dispatching mechanism is defined below as the function dsp which selects the expression to which $\hat{x}$ reduces:

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

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

The dispatching mechanism inspects a variation from left to right to find the first goal G satisfied by C, under a substitution θ that binds the variables of G. If this search succeeds, the results are the corresponding expression e and θ. Then $\hat{x}$ reduces to $e\,\theta$, i.e. to e whose variables are bound by θ. Instead, if the dispatching matching fails because no goal holds, the computation gets stuck because the program cannot adapt to the current context. Here, the dispatching mechanism has a fixed evaluation order, that can be made more flexible by assigning weights to alternative pairs $G.e$, as briefly discussed in the conclusions. As an example of context-dependent binding consider the function `setMainWindow` defined in Section 4 and apply it to a value w. Assume the environment ρ binds the parameters `txt_label` and `vcanvas` as done in Section 4. Furthermore, assume that context C satisfies the goals $\leftarrow \neg$`only_speech`() and $\leftarrow$ `video`(`hd`), $\neg$`only_text`() but not $\leftarrow$ `video`(`low`), $\neg$`only_text`().

$$\begin{aligned} \rho \vdash C,\ &\texttt{setMainWindow}\, w \to^* \\ &C,\ \texttt{addComp}\, w\, \texttt{txt\_label}; \texttt{addComp}\, w\, \texttt{vcanvas}; \texttt{e} \to \\ &C,\ \texttt{addComp}\, w\, (\texttt{getLabel}()); \texttt{addComp}\, w\, \texttt{vcanvas}; \texttt{e} \to^* \\ &C,\ \texttt{addComp}\, w\, \texttt{vcanvas}; \texttt{e} \to \\ &C,\ \texttt{addComp}\, w\, (\texttt{getHDCanvas}()); \texttt{e} \to^*\ \ C,\ \texttt{e} \end{aligned}$$

In the computation from the second to the third configuration, we retrieve the binding for `txt_label` and apply dsp to C and ρ(`txt_label`)

$$dsp(C,\ \rho(\texttt{txt\_label})) =$$
$$dsp(C, \leftarrow \neg\texttt{only\_speech}().\texttt{getLabel}()) = (\texttt{getLabel}(), \emptyset).$$

The same happens in the step from the fourth to the fifth configuration: we apply dsp to C and ρ(`vcanvas`) obtaining `getHDCanvas`() and the empty substitution $\emptyset$.

The rules for $e_1 \cup e_2$ sequentially evaluate e_1 and e_2 until they reduce to behavioural variations (rules (APPEND1, 2)). Then, they are concatenated together by renaming bound variables to avoid name captures (rule (APPEND3)). As an example of behavioural variation concatenation, consider the function `addDefault` of Section 4. In the following computation we apply `addDefault` to $p = (x)\{G_1.c_1, G_2.x\}$ and to c_2 (c_1, c_2 constants):

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

The rules for behavioural variation application $\#(e_1, e_2)$ evaluate the sub-expressions until e_1 reduces to $(x)\{Va\}$ (rule (VAAPP1)) and e_2 to a value v (rule (VAAPP2)). Then the rule (VaApp3) invokes the dispatching mechanism to select the relevant expression e from which the computation proceeds after v replaces x. Also in this case the computation gets stuck, if the dispatching mechanism fails. As an example consider the function `getExhibitData` and apply it to unit. The computation is

$$\begin{aligned}\rho \vdash C,\ &\texttt{getExhibitData}\,() \to^* \\ &C,\ \texttt{getRemoteData}\,\#(u, ()) \to^* \\ &C,\ \texttt{getRemoteData}(\texttt{receiveData}\,n) \\ &\qquad (*\ n \text{ is returned by } \texttt{getChannel}\ *)\end{aligned}$$

If the context C satisfies the goal $\leftarrow \texttt{direct\_comm}()$, in the computation from the second to the third configuration, the dispatching mechanism selects the first expression of the behavioural variation u (the one bound to `url` in the body of the function `getExhibitData`).

6 Type and Effect System

We now associate an ML_{CoDa} expression with an annotated type and an abstraction, called *history expression*. During the verification phase the virtual machine uses this history expression to ensure that the dispatching mechanism will always succeed at run time. First, we briefly present history expressions and then the rules of our type and effect system.

6.1 History Expressions

History Expressions [14] 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 context at run time, i.e., asserting/retracting facts and asking if a goal holds, as well as how behavioural variations will be "resolved".

The syntax of history expressions is the following

$$\begin{aligned}H &::= \epsilon \mid h \mid \mu h.H \mid H_1 + H_2 \mid H_1 \cdot H_2 \mid \\ &\qquad tell\, F \mid retract\, F \mid \Delta \\ \Delta &::= ask\, G.H \otimes \Delta \mid fail\end{aligned}$$

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}; 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 mimicking our dispatching mechanism.

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 Figure 5. A transition $C, H \to C', H'$ means that H reduces to H' in the context C and yields the context C'. Most rules are similar to the ones of Bartoletti et al. [14], and below we briefly comment on them.

The rules for sequence $H_1 \cdot H_2$ reduces H_1 up it becomes ϵ and then the overall expression reduces to H_2. The recursion $\mu h.H$ reduces to the body H substituting $\mu h.H$ for the recursion variable h. The sum $H_1 + H_2$ non-deterministically reduces to the history expression resulted from the reduction of either H_1 or H_2. 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 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 that 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, representing that the dispatching mechanism fails.

As a matter of fact, history expressions can also be seen as specifying context-fee languages, e.g. $\mu h.(\epsilon + tell\ F \cdot h \cdot retract\ F)$ defines the language of balanced parenthesis.

Note that differently from Degano et al. [42] here we are not concerned about the *sequence* of actions carried out from the initial context, but in *which* contexts can be reached starting from it. For this reason, here the transition system has no labels and we introduce no language-theoretic semantics, as done by Degano et al. [42], although easily doable.

6.2 Types and Effects

We give in Figure 6 a logical presentation of a type and effect system for ML_{CoDa}. For keeping the formal development easy we only consider monomorphic types and effects, and we assume that our Datalog is typed, i.e. each predicate has a fixed arity and a type. Many papers exist on this topic, and one can follow, e.g., a light version of the type system proposed by Mycroft and O'Keefe [68]. From here onwards, we simply assume that there exists a Datalog typing function γ that given a goal G returns a list of pairs (x, type-of-x), for all the variables x of G (γ is used e.g. in the rule TVARIATION in Figure 6).

6.2.1 Type judgements

The rules of our type and effect systems have the usual type environment Γ binding the variables of an expression:

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

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

Additionally, we introduce the parameter environment K that maps a parameter $\hat{x}$ to a pair consisting of a type and an abstract variation Δ. The information in Δ is used to resolve the binding for $\hat{x}$ at run time. Formally:

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

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

Our typing judgements have the form $\Gamma;\ K \vdash e : \tau \triangleright H$, expressing that in the environments Γ and K the expression e has type τ and effect H.

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

$$\frac{C, H_2 \to C',\, H_2'}{C, H_1 + H_2 \to C',\, H_2'} \qquad \frac{}{C,\, tell\, F \to C \cup \{F\},\, \epsilon} \qquad \frac{}{C,\, retract\, F \to C \backslash \{F\},\, \epsilon} \qquad \frac{C \vDash G}{C,\, ask\, G.H \otimes \Delta \to C,\, H}$$

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

Figure 5. Semantics of History Expressions

The syntax of types is

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

We have basic types (int, $bool$, $unit$), functional types, behavioural variations types, and facts. Some types are annotated 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 run time (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. Here, K stores the types and the abstract variations of parameters occurring inside the body of f. The history expression H is the latent effect of f, i.e. the sequence of actions which may be performed over the context during the function evaluation. Analogously, in the type $\tau_1 \xRightarrow{K|\Delta} \tau_2$ associated with the behavioural variation $bv = (x)\{Va\}$, K is a precondition for applying bv and Δ is an abstract variation representing the information that the dispatching mechanism uses at run time 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 $fail \otimes \Delta = \Delta$, so Δ_2 has a single trailing term $fail$); $K_1 \sqsubseteq K_2$ iff $(\,(\hat{x},\, \tau_1,\, \Delta_1) \in K_1$ implies $(\hat{x},\, \tau_2,\, \Delta_2) \in K_2 \wedge \tau_1 \leq \tau_2 \wedge \Delta_1 \sqsubseteq \Delta_2\,)$.

6.2.2 Typing rules

The most interesting rules of our type and effect system are in Figure 6. All the rules are collected in Appendix A. A few comments are in order.

We have rules for subtyping and subeffecting (Figure 6, top). As expected these rules 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 contravariant 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 nondeterministic summation of $tell\, F/retract\, F$ where F is any of the facts in the type of e. For example, consider the body of the function `setAccessibilityOpt` of Section 4. We know that the function `getCheckedOption` (call it e_1) returns either fact `user_acc_opt(deaf)` (call it F_1) or `user_acc_opt(blind)` (call it F_2). Now, call e_2 the application of e_1 to `accRadioButton`. The type of e_2 is $fact_{\{F_1,\, F_2\}}$, and its effect is H, which is the latent effect of e_1. So the overall type of `tell`(e_2) will be $unit$ and its effect $H \cdot (tell\, F_1 + tell\, F_2)$.

Rule (TPAR) looks for the type and the effect of the parameter $\hat{x}$ in the environment K. For example, consider what happens when type-checking the application `addComp window vcanvas` in the body of function `setMainWindow` in Section 4. Assume that `addComp` has type `window_t` $\xrightarrow{H_a}$ `component_t` $\xrightarrow{H_b}$ $unit$ and empty effect, `window` has type `component_t` and empty effect and that the binding for `vcanvas` in K is (`component_t`, Δ) where

$$\Delta = ask \leftarrow \texttt{video(hd)}, \neg\texttt{only\_text}().\ H_1 \otimes$$
$$ask \leftarrow \texttt{video(low)}, \neg\texttt{only\_text}().\ H_2 \otimes fail.$$

The application `addComp window vcanvas` thus has type $unit$ and effect $\Delta \cdot H_a \cdot H_b$.

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. For example, consider the behavioural variation defined in the body of function `getExhibitData` of Section 4 (call it bv_1). Assume that the unused argument _ has type $unit$ and that the two cases of this behavioural variation have type τ and effect H_1 and H_2, respectively, under the environment $\Gamma, \_ : unit$ (goals have no variables) and the guessed environment K'. Hence, the type of bv_1 will be $unit \xRightarrow{K'|\Delta} \tau$ with $\Delta = ask\, \texttt{direct\_comm}().H_1 \otimes ask\, \texttt{use\_qr\_code}(), \texttt{camera(on)}.H_2 \otimes fail$ and the effect will be empty.

The rule (TVAPP) type-checks behavioural variation applications and reveals the role of preconditions. As expected,

$$\text{(SREFL)}\ \frac{}{\tau \leq \tau} \qquad \text{(SFACT)}\ \frac{\phi \subseteq \phi'}{fact_{\phi} \leq fact_{\phi'}} \qquad \text{(SFUN)}\ \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad K \sqsubseteq K' \quad H \sqsubseteq H'}{\tau_1 \xrightarrow{K|H} \tau_2 \leq \tau_1' \xrightarrow{K'|H'} \tau_2'}$$

$$\text{(SVA)}\ \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad K \sqsubseteq K' \quad \Delta \sqsubseteq \Delta'}{\tau_1 \xRightarrow{K|\Delta} \tau_2 \leq \tau_1' \xRightarrow{K'|\Delta'} \tau_2'} \qquad \text{(TSUB)}\ \frac{\Gamma;\, K \vdash e \,:\, \tau' \triangleright H' \quad \tau' \leq \tau \quad H' \sqsubseteq H}{\Gamma;\, K \vdash e \,:\, \tau \triangleright H} \qquad \text{(TFACT)}\ \frac{}{\Gamma;\, K \vdash F \,:\, fact_{\{F\}} \triangleright \epsilon}$$

$$\text{(TTELL)}\ \frac{\Gamma;\, K \vdash e \,:\, fact_{\phi} \triangleright H}{\Gamma;\, K \vdash tell(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} tell\, F\right)} \qquad \text{(TRETRACT)}\ \frac{\Gamma;\, K \vdash e \,:\, fact_{\phi} \triangleright H}{\Gamma;\, K \vdash retract(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} retract\, F\right)} \qquad \text{(TPAR)}\ \frac{K(\hat{x}) = (\tau,\, \Delta)}{\Gamma;\, K \vdash \hat{x} \,:\, \tau \triangleright \Delta}$$

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

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

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

$$\text{(TDLET)}\ \frac{\Gamma, \overrightarrow{y} : \overrightarrow{\tau};\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1 \qquad \Gamma;\, K, (\hat{x},\, \tau_1,\, \Delta') \vdash e_2 \,:\, \tau_2 \triangleright H_2}{\Gamma;\, K \vdash dlet\, \hat{x} \,=\, e_1\, when\, G\, in\, e_2 \,:\, \tau_2 \triangleright H_2}$$

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

Figure 6. Typing rules for new constructs

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. For example, consider bv_1 above, that has type $unit \xRightarrow{K'|\Delta} \tau$. Assume to have the environments Γ and K, under which we wish to type-check the expression $\#(bv_1,\, ())$. If $K' \sqsubseteq K$, its type is τ and its effect is $ask\, \texttt{direct\_comm}().H_1 \otimes ask\, \texttt{use\_qr\_code}(), \texttt{camera}(\texttt{on}).H_2 \otimes fail$.

The rule (TAPPEND) asserts that two expressions e_1,e_2 with the same type τ, except for the abstract variations Δ_1, Δ_2 in their annotations, and effects H_1 and H_2, are combined into $e_1 \cup e_2$ with type τ, and concatenated annotations and effects. More precisely, the resulting annotation has the same precondition of e_1 and e_2 and abstract variation $\Delta_1 \otimes \Delta_2$, and effect $H_1 \cdot H_2$. For example, consider again the above bv_1; its type $int \xRightarrow{K'|\Delta} \tau$; the body of the function `addDefault` of Section 4. Let $bv_2 = (w)\{\texttt{True}.y\}$, and let its type be $unit \xRightarrow{K'|\Delta'} \tau$ and its effect be H_2. Then the type of $bv_1 \cup bv_2$ is $unit \xRightarrow{K'|\Delta \otimes \Delta'} \tau$ and the effect is H_2. The type of `addDefault` is $unit \xRightarrow{K'|\Delta} \tau \to \tau \to unit \xRightarrow{K'|\Delta \otimes \Delta'} \tau$.

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 for parameter $\hat{x}$. The type and the effect for the overall *dlet* expression are those of e_2.

6.2.3 Formal results

Our type and effect system is sound with respect to the operational semantics. To concisely state our results, it is convenient to introduce the following technical definition.

Definition 6.1 (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 $\forall \hat{x} \in dom(\rho)\,.\ \rho(x) = G_1.e_1, \ldots, G_n.e_n\ K(\hat{x}) = (\tau,\, \Delta)$ and $\forall i \in \{1, \ldots, n\}\,.\ \gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}\ \Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K \vdash e_i \,:\, \tau' \triangleright H_i$ and $\tau' \leq \tau$ and $\bigotimes_{i \in \{1, \ldots, n\}} G_i.H_i \sqsubseteq \Delta$.

The soundness of our type and effect system easily derive from the following results (the proofs are in Appendix A).

Theorem 6.1 (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 $H \sqsubseteq H'_s$.*

The Progress Theorem below assumes that the effect H *does not* reach *fail*, i.e. that the dispatching mechanism succeeds at run time. We take care of ensuring this property in Section 7. Hereafter we write $\rho \vdash C, e \not\to$ to intend that there exist no C' and e' such that $\rho \vdash C, e \to C', e'$; similarly $C, H_s \not\to^+ C', fail$ means that no computation of H in C reaches a failure.

Theorem 6.2 (Progress). *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho : K$. If $\rho \vdash C, e_s \not\to \wedge C, H_s \not\to^+ C', fail$ then e_s is a value.*

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.

Proposition 6.3 (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 6.4 (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 $C, H_s \not\to^+ C', fail$. Then either the computation of e_s terminates yielding a value $(\rho \vdash C, e_s \to^* C'', v)$ or it diverges, but it never gets stuck.*

We defined an inference algorithm for ML_{CoDa} [39]. We followed the approach proposed by Tang and Jouvelot [86], whose idea is to carry out the inference in two steps: the first computes the type of an expression e with no annotation, the second one reconstructs the annotations and the effect of e.

7 LOAD TIME ANALYSIS

As noted at the end of Section 4, an adaptive application may fail when executed in an environment not considered at design time, e.g. not offering the required resources. Our load time analysis aims at detecting this new class of run time errors.

In the execution model of ML_{CoDa} the compiler produces a triple (C_p, e_p, H_p) made of the application context, the object code and an effect over-approximating how the application e_p interacts with the context (see Section 3). Using this triple, the virtual machine of ML_{CoDa} performs a linking and verification phases at load 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 two are not contradictory. Still, the application is "open" with respect to its parameters. This calls for the verification phase: we check whether the application adapts to all evolutions of C that may occur at run time, i.e., that all dispatching invocations will always succeed. Only programs which pass this verification phase will be run. To do that efficiently and to pave the way for checking further properties (see the conclusions), we build a graph $\mathcal{G}$ describing the possible evolutions of the initial context, exploiting the history expression H_p. Technically, we compute $\mathcal{G}$ through a static analysis, specified in terms of Flow Logic [71], a declarative approach borrowing from and integrating many classical static techniques [36], [48], [75], [61]. The distinctive feature of Flow Logic is to separate the *specification* of the analysis from its actual *computation*. Intuitively, the specification describes when its results, namely analysis *estimates*, are *acceptable*, i.e. sound with respect to the dynamic semantics. Formally, a specification consists of a set of clauses defining the acceptability relationship of estimates. Furthermore, Flow Logic provides us with a methodology to define a correct and efficient analysis algorithm, by reducing the specification to a constraint satisfaction problem. Below, we specify our analysis in a logical form and then we define an algorithm for it.

7.1 Specification of the Analysis

Here, we introduce our analysis of history expression and define the notion of validity on them. Intuitively, a history expression is valid for an initial context if the dispatching mechanism always succeeds.

To support the formal development, we assume that history expressions are labelled from a given set of Lab. Actually, the labels can be mechanically attached, in injection with the nodes in the abstract syntax tree of H. Formally:

$$\begin{aligned} H ::= & \ \text{ɜ} \mid \epsilon^l \mid h^l \mid (\mu h.H)^l \mid tell\, F^l \mid retract\, F^l \mid \\ & (H_1 + H_2)^l \mid (H_1 \cdot H_2)^l \mid \Delta \\ \Delta ::= & (ask\, G.H \otimes \Delta)^l \mid fail^l \end{aligned}$$

We introduce for technical reasons a new empty history expression ɜ 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 (ɜ). The semantics of history expressions is accordingly modified, and Figure 7 displays it (apart from labels and ɜ, this semantics and the old one clearly coincide). Furthermore, without loss of generality, we assume that all the bound variables occurring in a history expression are distinct. To keep trace of the history expression $(\mu h.H_1^{l_1})^{l_2}$ where the a bound variable h^l is introduced, we shall use a suitable function, called $\mathbb{K}$.

A result of the analysis is a pair of functions $\Sigma_\circ, \Sigma_\bullet : Lab \to \wp(Context \cup \{\divideontimes\})$ where $\divideontimes$ is the distinguished "failure" context representing a dispatching failure. For each label l, the set $\Sigma_\circ(l)$ over-approximates the set of contexts that may arise before evaluating H^l (call it *pre-set*); $\Sigma_\bullet(l)$ over-approximates the set of contexts that may result from the evaluation of H^l (call it *post-set*).

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

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

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

Figure 7. New semantics of History Expressions

We define the specification of our analysis through the validity relation

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

where $\mathcal{AE} = (Lab \to \wp(Context \cup \{\divideontimes\}))^2$ 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 7.2 to check whether H, hence the expression e it is an abstraction of, will never fail in a given initial context C.

In Figure 8 we give the set of inference rules that inductively define the validity relation $\vDash$. Now, we briefly comment on them. In the description below, we denote with $\mathcal{E}$ the estimate $(\Sigma_\circ, \Sigma_\bullet)$, and we omit immaterial labels.

The rule (ANIL) says that every pair of functions is an acceptable estimate for the semantic empty history expression $\mathfrak{z}$. 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 $H = (\mathfrak{z} \cdot H_1^{l_2})^l$ 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 $\mathbb{K}(h)$.

7.1.1 Validity and Viability

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

Definition 7.1 (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 following theorems state the correctness of our approach. The first guarantees that there exists a minimal valid analysis estimate. Its existence is proved by showing that the set of acceptable analyses forms a Moore family [71].

Theorem 7.1 (Existence of solutions)**.** *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

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

Theorem 7.2 (Subject Reduction)**.** *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$.*
If for all $C \in \Sigma_\circ(l)$ such that $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H'^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.

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 7.2 (Viability)**.** Let H_p be an 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 $\forall l \in dom(\Sigma_\bullet)\backslash lfail(H_p),\ \divideontimes \notin \Sigma_\bullet(l)$.

Below we illustrate how viability is checked using a couple of examples. Consider the following history expression H_a

$$((tell\, F_1^1 \cdot (retract\, F_2^2 \cdot (ask\, F_8.retract\, F_5^3 \otimes fail^4)^5)^6)^7 + (ask\, F_5.retract\, F_8^8 \otimes (ask\, F_3.retract\, F_4^9 \otimes fail^{10})^{11})^{12})^{13}$$

and the initial context $C = \{F_2,\ F_5,\ F_8\}$, consisting of facts only. For each label l occurring in H_a, Figure 9 shows the corresponding values of $\Sigma_\circ^1(l)$ and $\Sigma_\bullet^1(l)$, respectively. Since $\Sigma_\bullet^1$ does not contain $\divideontimes$, H_a is viable for C.

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

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

$$\text{(ASEQ2)}\ \frac{(\Sigma_\circ,\Sigma_\bullet)\vDash H_2^{l_2}\qquad\Sigma_\circ(l)\subseteq\Sigma_\bullet(l_2)\qquad\Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet)\vDash (\mathfrak{z}\cdot H_2^{l_2})^l} \qquad \text{(ASUM)}\ \frac{\begin{array}{l}(\Sigma_\circ,\Sigma_\bullet)\vDash H_1^{l_1}\qquad\Sigma_\circ(l)\subseteq\Sigma_\circ(l_1)\qquad\Sigma_\bullet(l_1)\subseteq\Sigma_\bullet(l)\\(\Sigma_\circ,\Sigma_\bullet)\vDash H_2^{l_2}\qquad\Sigma_\circ(l)\subseteq\Sigma_\circ(l_2)\qquad\Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l)\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\vDash (H_1^{l_1}+H_2^{l_2})^l}$$

$$\text{(AASK1)}\ \frac{\forall C\in\Sigma_\circ(l)\qquad\begin{array}{l}(C\vDash G\implies(\Sigma_\circ,\Sigma_\bullet)\vDash H^{l_1}\qquad\Sigma_\circ(l)\subseteq\Sigma_\circ(l_1)\qquad\Sigma_\bullet(l_1)\subseteq\Sigma_\bullet(l))\\(C\nvDash G\implies(\Sigma_\circ,\Sigma_\bullet)\vDash \Delta^{l_2}\qquad\Sigma_\circ(l)\subseteq\Sigma_\circ(l_2)\qquad\Sigma_\bullet(l_2)\subseteq\Sigma_\bullet(l))\end{array}}{(\Sigma_\circ,\Sigma_\bullet)\vDash (askG.H^{l_1}\otimes\Delta^{l_2})^l} \qquad \text{(AASK2)}\ \frac{\divideontimes\in\Sigma_\bullet(l)}{(\Sigma_\circ,\Sigma_\bullet)\vDash fail^l}$$

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

Figure 8. Specification of the analysis for History Expressions

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

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

Indeed H'_a is not viable because the goal F_3 does not hold in C, and this is reflected by the occurrences of $\divideontimes$ in $\Sigma_\bullet^2(4)$ and $\Sigma_\bullet^2(10)$ as shown in Figure 10.

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 run time. The virtual machine can use $\mathcal{G}$ to study how the application interacts with and affects the context.

In the following let $Fact^*$ and Lab^* 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 an arc between two nodes C_1 and C_2 if C_2 is obtained from C_1 through telling or removing a fact F.

Definition 7.3 (Evolution graph)**.** Let H be a history expression, C be an initial context, and $(\Sigma_\circ,\Sigma_\bullet)$ be a valid analysis estimate.
The evolution graph of C is $\mathcal{G}=(N,A)$, where

$$\begin{aligned}N &= \bigcup_{l\in Lab^*}(\Sigma_\circ(l)\cup\Sigma_\bullet(l))\\ A &= \{(C_1,C_2)\mid \exists F\in Fact^*,\ l\in Lab^* \text{ such that}\\ &\qquad C_1\in\Sigma_\circ(l)\wedge C_2\in\Sigma_\bullet(l)\wedge\\ &\qquad((C_1=C_2\setminus\{F\})\vee(C_2=C_1\setminus\{F\})\vee(C_2=\divideontimes))\}\end{aligned}$$

As examples of evolution graphs consider the context C and the history expressions H_a and H'_a introduced above. Figure 9 depicts the evolution graph of C for H_a, where the node $\divideontimes$ is not reachable, showing H_a viable for C. Instead, in the evolution graph of C for H'_a, displayed in Figure 10, the node $\divideontimes$ is reachable, showing H'_a not viable for C (for readability, we labelled arcs with the performed actions).

Here we simply exploit evolution graphs to detect adaptation failures, and if there is one, the application is prevented to run. A more advanced usage is to visit the graph and single out which behavioural variation will never rise adaptation errors. All the other behavioural variation can be instrumented to contain a call to suitable recovery mechanisms. Other properties can also be checked on evolution graphs, among which security policies to enforce [17], or properties concerning a correct usage of contextual resources.

7.2 Analysis Algorithm

The idea underlying the algorithm that computes the valid estimates consists in reformulating the analysis specification as a constraint satisfaction problem: its minimal solution yields the minimal valid analysis estimate.

7.2.1 Constraint generation

Given a history expression H we generate constraints of the form $E\subseteq X\in\mathcal{SC}$ where E is a *set-expression* and X a *variable*, both denoting sets of contexts. Intuitively, the set denoted by E is constrained to be a subset of the set X.

To formalise the analysis as a constraint satisfaction problem, we first define the syntax of set-expressions and their semantics (Definition 7.4); then we introduce the function $\mathcal{C}[\_]:\mathbb{H}\to\mathcal{SC}$ that generates constraints from a history expression (Definition 7.5); furthermore we define the constraints satisfaction relation $\vDash_{sc}\subseteq\mathcal{AE}\times\mathcal{SC}$ saying

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

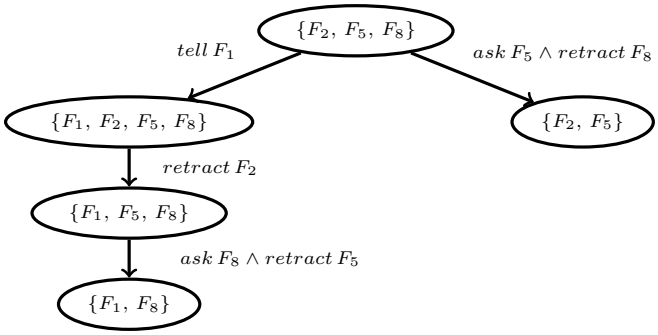

Figure 9. The analysis result (top) and the evolution graph (bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression H_a.

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

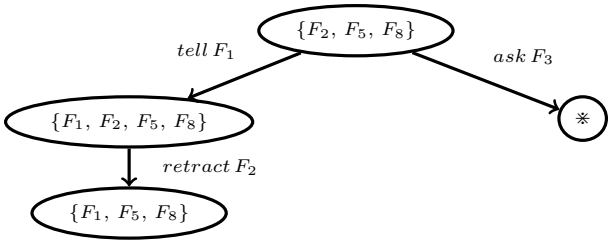

Figure 10. The analysis result (top) and the evolution graph (bottom) for the context $C = \{F_2, F_5, F_8\}$ and the history expression H'_a

when a set of constraints is satisfied by an analysis estimate (Definition 7.6); finally, we prove that the valid estimates of the analysis (Definition 7.1) coincide with the solutions of the constraint system (Theorem 7.3).

Let H be the history expression to be analysed, and let $Goal^*$, $Fact^*$ and Lab^* be the goals, the facts and the labels occurring in H, respectively; furthermore, let $Context^*$ be the set of all contexts that may be generated from the initial context C by asserting and retracting the facts in $Fact^*$. Note that all the sets above are finite. Our set-expressions are defined as

$$X \in SetVar \quad C \in Context^* \quad G \in Goals^* \quad F \in Facts^*$$
$$E ::= X \mid \{\ast\} \mid \{C\} \mid E \sqcup F \mid E \setminus F \mid$$
$$E \vDash G \mid E \nvDash G \mid E_1 \Rightarrow E_2$$

where X is a variable; $\{\ast\}$ is the singleton containing the failure context; the expression $E \sqcup F$ ($E \setminus F$, respectively) denotes the set of contexts of E where we have added (removed, respectively) the fact F to each element; the expression $E \vDash G$ ($E \nvDash G$, respectively) denotes the subset of E containing only the contexts satisfying (not satisfying, respectively) the goal G; the expression $E_1 \Rightarrow E_2$ is a conditional expression: intuitively, the result is either empty if the set E_1 is such, otherwise it is E_2.

The idea underlying our representation is based on the fact that the analysis is syntax-driven. Consequently, the labels relevant for computing the estimates of the analysis are those in Lab^*, and the contexts occurring therein are those belonging to $Context^*$. Since Lab^* is finite, we can represent a function $\Sigma\colon Lab^* \to \wp(Context^* \cup \{\ast\})$ through a set of variables. The value of each variable is intended to be the set of contexts associated with a given label. To make this link manifest, we define $SetVar$ as

$$SetVar = \{\, \widehat{\Sigma}_\circ(l) \mid l \in Lab^* \,\} \cup \{\, \widehat{\Sigma}_\bullet(l) \mid l \in Lab^* \,\}$$

For the sake of clarity, we also subscript variables with $\circ$ and $\bullet$ to indicate to which element of analysis estimate the variable refers.

The meaning of a set-expression is formalised as follows.

Definition 7.4 (Set-expression semantics)**.** Given an analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ the semantics of set expressions is given by the function $[\![\_]\!]\colon E \to \mathcal{AE} \to \wp(Context^* \cup \{\ast\})$ defined as

$$[\![\widehat{\Sigma}_\circ(l)]\!](\Sigma_\circ, \Sigma_\bullet) = \Sigma_\circ(l)$$
$$[\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ, \Sigma_\bullet) = \Sigma_\bullet(l)$$
$$[\![\{\ast\}]\!](\Sigma_\circ, \Sigma_\bullet) = \{\ast\}$$
$$[\![\{C\}]\!](\Sigma_\circ, \Sigma_\bullet) = \{C\}$$
$$[\![E \sqcup F]\!](\Sigma_\circ, \Sigma_\bullet) = \{C \cup \{F\} \mid C \in [\![E]\!](\Sigma_\circ, \Sigma_\bullet)\}$$
$$[\![E \setminus F]\!](\Sigma_\circ, \Sigma_\bullet) = \{C \setminus \{F\} \mid C \in [\![E]\!](\Sigma_\circ, \Sigma_\bullet)\}$$
$$[\![E \vDash G]\!](\Sigma_\circ, \Sigma_\bullet) = \{C \in [\![E]\!](\Sigma_\circ, \Sigma_\bullet) \mid C \vDash G\}$$
$$[\![E \nvDash G]\!](\Sigma_\circ, \Sigma_\bullet) = \{C \in [\![E]\!](\Sigma_\circ, \Sigma_\bullet) \mid C \nvDash G\}$$
$$[\![E_1 \Rightarrow E_2]\!](\Sigma_\circ, \Sigma_\bullet) = \begin{cases} [\![E_2]\!](\Sigma_\circ, \Sigma_\bullet) & \text{if } [\![E_1]\!](\Sigma_\circ, \Sigma_\bullet) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Given a history expression H, the function $\mathcal{C}[\_]\colon \mathbb{H} \to \mathcal{SC}$ generates the desired set of constraints, mimicking the specification rules in Figure 8.

Definition 7.5 (Constraints generation)**.** Given a history expression H^l and an initial context C, the set of constraints

for H^l and C is $\mathcal{S} = \{\{C\} \subseteq \widehat{\Sigma}_\circ(l)\} \cup \mathcal{C}\left[H^l\right]$, where the function $\mathcal{C}\left[\_\right] : \mathbb{H} \to \mathcal{SC}$ is inductively defined as follow

$$\mathcal{C}\left[\nexists\right] = \emptyset$$
$$\mathcal{C}\left[\epsilon^l\right] = \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[tell\, F^l\right] = \{\widehat{\Sigma}_\circ(l) \sqcup F \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[retract\, F^l\right] = \{\widehat{\Sigma}_\circ(l) \setminus F \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[(H_1^{l_1} \cdot H_2^{l_2})^l\right] = \mathcal{C}\left[H_1\right] \cup \mathcal{C}\left[H_2\right] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1),\, \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[(H_1^{l_1} + H_2^{l_2})^l\right] = \mathcal{C}\left[H_1\right] \cup \mathcal{C}\left[H_2\right] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l),\ \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[(\mu h.H^{l_1})^l\right] = \mathcal{C}\left[H\right] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[h^l\right] = \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\} \quad \mathbb{K}(h) = (\mu h.H)^{l_1}$$
$$\mathcal{C}\left[(askG.H^{l_1} \otimes \Delta^{l_2})^l\right] = \mathcal{C}\left[H\right] \cup \mathcal{C}\left[\Delta\right] \cup \{\widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1),\ \widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l),\ \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2),\ \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$$
$$\mathcal{C}\left[fail^l\right] = \{\divideontimes\} \subseteq \widehat{\Sigma}_\bullet(l)$$

In the following, it is convenient to assume that $\mathcal{C}\left[H^l\right]$ also includes $\{C\} \subseteq \widehat{\Sigma}_\circ(l)$.

By using the semantics of set-expressions, we define the relationship $\vDash_{sc} \subseteq \mathcal{AE} \times \mathcal{SC}$ specifying when an analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ satisfies a set of constraints

Definition 7.6 (Constraints satisfaction)**.** Given an analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$ and a set of constraints $sc \in \mathcal{SC}$, the relation $\vDash_{sc}$ is defined as

$$(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} sc \iff \forall E_1 \subseteq E_2 \in sc \quad [\![E_1]\!](\Sigma_\circ, \Sigma_\bullet) \subseteq [\![E_2]\!](\Sigma_\circ, \Sigma_\bullet)$$

The following theorem ensures that the formulations of the analysis given in Definitions 7.1 and 7.6, are equivalent, i.e. the solutions of the set constraints $\mathcal{C}\left[H\right]$ are valid analysis estimates and vice versa.

Theorem 7.3. *Let H be a history expression and let $(\Sigma_\circ, \Sigma_\bullet)$ be an analysis estimate, then*

$$(\Sigma_\circ, \Sigma_\bullet) \vDash H \iff (\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}\left[H\right]$$

As an example, consider the history expression $H_p = (tell\, F_1^1 \cdot retract\, F_2^2)^3$ and the initial context $C = \{F_2, F_3, F_5\}$, made of facts only. The table at the top of Figure 11 shows the constraints generated for each subterm of H_p, whose union gives the constraints for H_p. For the subterm $tell\, F_1^1$ we generate $\widehat{\Sigma}_\circ(1) \sqcup F_1$, the set of contexts where the fact F_1 is added to each element of $\widehat{\Sigma}_\circ(1)$, so mimicking the premise of the rule (ATELL). Analogously, for the subterm $retract\, F_2^2$: the constraint $\widehat{\Sigma}_\circ(2) \setminus F_2$ records that the fact F_2 is removed from each element of $\widehat{\Sigma}_\circ(2)$) (see the premise of the rule (ARETRACT)). The constraints in the last row correspond to the preconditions of the rule (ASEQ1). Additionally, they include the constraint $\{F_2, F_3, F_5\} \in \widehat{\Sigma}_\circ(3)$ as required by the definition of valid estimate (Definition 7.1). The valid analysis estimate displayed at the bottom of Figure 11 is a solution to the constraints for H_p.

7.2.2 Constraint solution

To solve a system of constraints we define the worklist algorithm in Figure 12, by instantiating the general schema of Nielson et al. [70] (Chapter 6). Given a set of constraints $\mathcal{S}$, it produces as solution an assignment $\mathcal{E}$ (represented as an array) for the variables occurring in the constraints. The algorithm uses three data structures: the list $\mathcal{W}$ which records the constraints to be further elaborated; the array $\mathcal{E}$, indexed by variables, which represents the current (partial) solution; and the array $\mathcal{A}$ which stores for each variable which are the constraints its value influences.

In the first step we initialise our data structures. At the end of this step $\mathcal{W}$ stores the constraint $\{C\} \subseteq X$, put on the initial context C; $\mathcal{E}$ gets the value $\emptyset$ for each element; each element X of $\mathcal{A}$ contains the constraints where X occurs in the left-hand side.

In the second step of the algorithm we compute the solution by stepwise refining $\mathcal{E}$; at the end of the execution $\mathcal{E}$ stores the minimal solution of the constraints in $\mathcal{S}$. In each iteration we extract a constraint $E \subseteq X$ from $\mathcal{W}$ and compute the value of E (stored in `new`) in the current solution $\mathcal{E}$ through the semantic function $[\![\_]\!]$. Note that for readability we assume an implicit type coercion from the array $\mathcal{E}$ to an analysis estimate $(\Sigma_\circ, \Sigma_\bullet)$. If the value `new` is not included in the assignment for the variable X, i.e. the constraint is not satisfied, we update its value by adding the one of E ($\mathcal{E}[X] \cup$ `new`). Changing the value for variable X may affect the satisfiability of the constraints $E' \subseteq X'$, in which X occurs in E'. For this reason, we add to $\mathcal{W}$ all the constraints $\mathcal{A}$[X]. The algorithm terminates when $\mathcal{W}$ is empty, i.e. when all the constraints are satisfied.

Figure 13 shows the iterations of the algorithm to solve the constraints in Figure 11. After the initialisation (iteration 0), $\mathcal{W}$ contains the constraint $\{C\} \subseteq \widehat{\Sigma}_\circ(3)$, the array $\mathcal{E}$ gets $\emptyset$ for all elements and $\mathcal{A}$ is initialised as shown at the bottom of Figure 13. After the iteration 1, we have that $\{C\} \subseteq \widehat{\Sigma}_\circ(3)$ is removed from $\mathcal{W}$, $\mathcal{E}[\widehat{\Sigma}_\circ(3)]$ includes the context C and the constraint $\widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1)$ is inserted into $\mathcal{W}$. The algorithm terminates at the iteration 6 when $\mathcal{W}$ becomes empty.

Our algorithm inherits the correctness and the properties from the general schema, as stated by the following theorem:

Theorem 7.4. *Let H be a history expression of size n and let h be the height of the complete lattice $\wp(Context^* \cup \{\divideontimes\})$. The algorithm in Figure 12 terminates and computes the minimal solution of the constraints $\mathcal{C}\left[H\right]$ in time $O(h \cdot n)$.*

The complexity of our algorithm depends on the value of height h of $\wp(Context^* \cup \{\divideontimes\})$. We conjecture that h is not a constant, and that it can instead be bound from above by a function in the size n of the history expression.

Subterms of H_p	Constraints
$tell\ F_1^1$	$\{\ \widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1)\ \}$
$retract\ F_2^2$	$\{\ \widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2)\ \}$
$(tell\ F_1^1 \cdot retract\ F_2^2)^3$	$\{\ \widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1), \widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2),$ $\widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3), \{F_2, F_3, F_5\} \in \widehat{\Sigma}_\circ(3)\ \}$

	1	2	3
$\Sigma_\circ$	$\{\{F_2, F_3, F_5\}\}$	$\{\{F_1, F_2, F_3, F_5\}\}$	$\{\{F_2, F_3, F_5\}\}$
$\Sigma_\bullet$	$\{\{F_1, F_2, F_3, F_5\}\}$	$\{\{F_1, F_3, F_5\}\}$	$\{\{F_1, F_3, F_5\}\}$

Figure 11. The constraints (on top) and their solution (on bottom) for the history expression $H_p = (tell\ F_1^1 \cdot retract\ F_2^2)^3$ and the context $C = \{F_2, F_3, F_5\}$.

Input: A set $\mathcal{S}$ of constraints $\mathtt{E_1} \subseteq \mathtt{X_1}, \ldots, \mathtt{E_n} \subseteq \mathtt{X_n}$
Output: The least solution $\mathcal{E}$

Step 1: Initialization of $\mathcal{W}$, $\mathcal{E}$ and $\mathcal{A}$;

$\mathcal{W} := \emptyset$;
for $\mathtt{X_i}$ **do** $\mathcal{A}[\mathtt{X_i}] = \emptyset$;
;

for $\mathtt{E} \subseteq \mathtt{X} \in \mathcal{S}$ **do**
  $\mathcal{E}[\mathtt{X}] := \emptyset$;
  for $\mathtt{X}' \in vars(\mathtt{E})$ **do**
    $\mathcal{A}[\mathtt{X}'] := \mathcal{A}[\mathtt{X}'] \cup \{\mathtt{E} \subseteq \mathtt{X}\}$;
  end
  if $\mathtt{E} = \{C\}$ **then**
    $\mathcal{W} := \mathcal{W} \cup \{\mathtt{E} \subseteq \mathtt{X}\}$;
end

Step 2: Iteration (updating W and $\mathcal{E}$);

while $\mathcal{W} = \{\mathtt{E} \subseteq \mathtt{X}\} \cup \mathcal{W}'$ **do**
  $\mathcal{W} := \mathcal{W}'$;
  $\mathtt{new} := [\![\mathtt{E}]\!]\ \mathcal{E}$;
  if $\mathtt{new} \not\subseteq \mathcal{E}[\mathtt{X}]$ **then**
    $\mathcal{E}[\mathtt{X}] := \mathcal{E}[\mathtt{X}] \cup \mathtt{new}$;
    for $\mathtt{E}' \subseteq \mathtt{X}' \in \mathcal{A}[\mathtt{X}]$ **do**
      $\mathcal{W} := \mathcal{W} \cup \{\mathtt{E}' \subseteq \mathtt{X}'\}$;
    end
end

Figure 12. The worklist algorithm to solve constraints over set-expression

If the value of h is large, the algorithm may perform many iterations before converging to the solution. However, several approaches have been proposed to keep efficient the execution of the algorithm in practice, e.g. based on widening operators [19].

8 A Prototypical Compiler

This section summarises the ideas underlying the ongoing implementation of our language as an extension of F#,[2] a dialect of ML. Presently, we only have preliminary results for the implementation of the two-step static analysis of Sections 6 and 7. The type system is under implementation, and we are using F# *type providers*; the implementation of the CFA is more standard. Instead, all the ML_{CoDa} constructs have been fully implemented.[3] The choice of F# alleviates the effort of re-implementing all the well-known constructs, mainly because we could exploit the metaprogramming facilities of F#, such as code introspection, quotation and reflection. Additionally, the F# compiler is stable and generates optimised bytecode; it is officially supported by Microsoft and fully integrated inside the .NET environment [4] (and Mono,[5] its open-source counterpart). F# applications can readily run on all platforms (computers or mobile devices) supported by CLR [20] (or Mono) and can exploit all the features provided, including a vast collection of libraries and modules, and also a just-in-time mechanism for compiling to native code. Since ML_{CoDa} is implemented as a standard .NET library, no modifications are needed either to the compiler or to the underlying run time. This is a plus because it lowers the complexity of deploying and maintaining applications; it preserves the compatibility with other language extensions. Finally, the learning cost is minimised for a user of a functional language like F# or ML, who only needs training on the specific aspects of adaptivity and possibly on Datalog.

The independence of the development of the context from that of the application is well supported by .NET through the notion of *assemblies*. They work just as modules and offer us a natural way to separate the code for the two components of ML_{CoDa}. In practice, a requirements engineer writes a bunch of Datalog sources that are ahead-of-time translated to .NET code using our compiler `ypc` built on the YieldProlog library.[6] In this way, the interaction and the data exchange between the application and the context is easy because the .NET type system is uniformly used everywhere. The problem of impedence mismatch [66] is thus completely avoided.

The application programmer instead writes F# code, annotating the functions which use ML_{CoDa} extensions with an attribute called `Coda.Code`. This code is compiled through the standard compiler `fsharpc` because the operations

2. http://fsharp.org

3. https://github.com/vslab/fscoda
4. http://www.microsoft.com/net
5. http://www.mono-project.com/
6. https://github.com/vslab/YieldProlog

Iteration	$\mathcal{W}$	$\widehat{\Sigma}_\circ(1)$	$\widehat{\Sigma}_\circ(2)$	$\widehat{\Sigma}_\circ(3)$	$\widehat{\Sigma}_\bullet(1)$	$\widehat{\Sigma}_\bullet(2)$	$\widehat{\Sigma}_\bullet(3)$
0	$\{\mathbf{C}\} \subseteq \widehat{\mathbf{\Sigma}}_\circ(\mathbf{3})$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	~~$\{C\} \subseteq \widehat{\Sigma}_\circ(3)$~~ $\widehat{\mathbf{\Sigma}}_\circ(\mathbf{3}) \subseteq \widehat{\mathbf{\Sigma}}_\circ(\mathbf{1})$	$\emptyset$	$\emptyset$	$\mathbf{C}$	$\emptyset$	$\emptyset$	$\emptyset$
2	~~$\widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1)$~~ $\widehat{\mathbf{\Sigma}}_\circ(\mathbf{1}) \sqcup \mathbf{F_1} \subseteq \widehat{\mathbf{\Sigma}}_\bullet(\mathbf{1})$	$\mathbf{C}$	$\emptyset$	C	$\emptyset$	$\emptyset$	$\emptyset$
3	~~$\widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1)$~~ $\widehat{\mathbf{\Sigma}}_\bullet(\mathbf{1}) \subseteq \widehat{\mathbf{\Sigma}}_\circ(\mathbf{2})$	C	$\emptyset$	C	$\mathbf{C_1}$	$\emptyset$	$\emptyset$
4	~~$\widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2)$~~ $\widehat{\mathbf{\Sigma}}_\circ(\mathbf{2}) \setminus \mathbf{F_2} \subseteq \widehat{\mathbf{\Sigma}}_\bullet(\mathbf{2})$	C	$\mathbf{C_1}$	C	C_1	$\emptyset$	$\emptyset$
5	~~$\widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2)$~~ $\widehat{\mathbf{\Sigma}}_\bullet(\mathbf{2}) \subseteq \widehat{\mathbf{\Sigma}}_\bullet(\mathbf{3})$	C	C_1	C	C_1	$\mathbf{C_2}$	$\emptyset$
6	~~$\widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3)$~~ $\emptyset$	C	C_1	C	C_1	C_2	$\mathbf{C_2}$

$$C = \{F_2, F_3, F_5\}, C_1 = \{F_1, F_2, F_3, F_5\}, C_2 = \{F_1, F_3, F_5\}$$

$\widehat{\Sigma}_\circ(1)$	$\widehat{\Sigma}_\circ(2)$	$\widehat{\Sigma}_\circ(3)$	$\widehat{\Sigma}_\bullet(1)$	$\widehat{\Sigma}_\bullet(2)$	$\widehat{\Sigma}_\bullet(3)$
$\widehat{\Sigma}_\circ(1) \sqcup F_1 \subseteq \widehat{\Sigma}_\bullet(1)$	$\widehat{\Sigma}_\circ(2) \setminus F_2 \subseteq \widehat{\Sigma}_\bullet(2)$	$\widehat{\Sigma}_\circ(3) \subseteq \widehat{\Sigma}_\circ(1)$	$\widehat{\Sigma}_\bullet(1) \subseteq \widehat{\Sigma}_\circ(2)$	$\widehat{\Sigma}_\bullet(2) \subseteq \widehat{\Sigma}_\bullet(3)$	$\emptyset$

Figure 13. The iterations of the worklist algorithm to solve constraints in Figure 11 (top) and the content of the corresponding array $\mathcal{A}$ (bottom).

needed to adapt the application to contexts are transparently handled by our run time support. In particular, ML_{CoDa}-specific constructs are just-in-time replaced by their F# implementation when they are about to run. This translation step is performed as a single pass. The just-in-time compiler identifies and picks up the quotations representing the source code in order to visit the abstract syntax tree. The generation of the new code is directly performed during the visit taking advantage of some tracking of the environment. Indeed, the ML_{CoDa} constructs cannot be implemented through macro-expansion because their translation depends on the scope chain and thus it requires keeping an additional symbol table for parameters.

We are currently experimenting on ML_{CoDa} and its implementation through concrete case studies; preliminary results are in Canciani et al. [26], which presents a simulator of an e-Healthcare system of a pediatric hospital, under development in cooperation with a small group of professionals. Besides evaluating the expressivity in handling contexts and in writing applications, we expect to experiment on the feasibility and effectiveness of our two-step analysis. We plan to implement additional case studies and to perform further benchmarks comparing our proposal with others in the literature [81], [5].

9 Concluding Remarks

We presented a language-based approach to adaptive programming, within the Context-Oriented Programming paradigm. Crucial to our proposal is the two-component core language ML_{CoDa}. The first component is specially designed for declaratively modelling the context. The second constituent extends a core of ML with high-level constructs for adaptation.

We also proposed a programming and an execution model. The programming model is characterised by a predefined API that we assume to be provided by the virtual machine. Our execution model is characterised by a linking and a verification phase. The linking phase enables the application to use the capabilities of the hosting system. The verification phase guarantees that the application will indeed adapt to the current context.

In more detail, our static verification technique for ML_{CoDa} consists of two phases: in the first, a type and effect system type-checks programs and computes a sound abstraction of their behaviour. The second phase occurs right after linking the application to the hosting system. This load time analysis uses the program abstraction to verify that the dispatching mechanism of the application will always succeed.

We have implemented ML_{CoDa} as an extension of F# and we are currently working on an efficient implementation of our two static analyses based on the type and effect inference algorithm [39]. Also, we have started experimenting on our language considering some applications.[7] Preliminary results confirm that the expressivity of Datalog helps in designing and handling the context in a flexible and succinct way: the deduction machinery of Datalog permits to compute the needed properties of the current context on need, avoiding the programmer to explicitly enumerate the relevant ones. Furthermore one can naturally write compact adaptive code in a functional style, by using our constructs. In particular contextual binding through `dlet`, and the capability of passing behavioural variations as parameters.

9.1 Discussion and Future Work

We briefly discuss below the main differences of ML_{CoDa} with respect to those proposals in the literature close to ours.

Almost all the calculi surveyed in Section 2 represent a context as a stack of active layers, while ours is a knowledge base, that offers primitives for easily storing and retrieving contextual data through Datalog queries. As noted in Section 4, solving these queries may require complex deductions, involving nested predicates and also API routines. While our dispatching mechanism checks properties of structured data, in most cases layers carry no data. However, Appeltauer et al. [5] discuss stateful layers, which are endowed with a notion of store.

In the literature behavioural variations are often implemented as partially defined methods, not first-class, as they are in ML_{CoDa}. The most notable exception is *ContextL* [35], that is based on *Common Lisp*, from which it inherits higher-order features.

Many of the above adaptive languages include a `proceed` construct, a sort of `super` invocation in object

7. https://github.com/vslab/fscoda

oriented languages [49], typically used for composing active behavioural variations. This construct is related to the idea of representing the context as a stack of layers, and it is unclear whether it makes sense to introduce a similar construct also in a full-fledged declarative context as ours. Nevertheless, one could add to ML_{CoDa} a construct similar to *call-next-method* [89], in order to run the next case with a goal satisfied, within the active behavioural variation.

In ML_{CoDa} a fact asserted through a `tell` holds until it is explicitly retracted. An alternative would be using the construct `with` that imposes a scope on the context modifications. Of course, scoping can be achieved by a disciplined use of `tell` and `retract`, with no changes to our static analysis. In a previous work of ours [42], we addressed this issue, and considered also the `without` construct, also from the static analyses viewpoint.

Our behavioural variation as lists of guarded expressions are close to *predicated generic functions* [89]. These functions are methods defined by cases that are selected on the basis of predicates implemented as LISP functions. Our dispatching mechanism is therefore similar to theirs, except that ours is guaranteed to always terminate.

We deal with the values of variables that change from a context to another through a notion of dynamic *context-dependent* binding. This problem has been faced by von Löwis at al. [91] who propose a mechanism of dynamic variables inside PyContext. These variables are Python objects, manipulated by special setter and getter methods, the value of which is set upon entering a context and restored while leaving it, in a stack-like way. In PyContext the scoping is kept static, and only the value bound to variables is dynamically determined. This differs from our approach because our parameters are dynamically scoped and need no special operations to be accessed. Tanter [87] proposes a more general solution in which values, not only variables, depend on the context. The mechanisms that manipulate contextual values can be implemented in a suitable library or directlty in the interpreter of the language. These features also allow scoping side-effects within a certain context. Our dynamic context-dependent binding can be implemented by Tanter's mechanisms, so it turns out to be a special case of his.

Other papers in the literature share our view of having distinct formalisms for specifying the context and the applications. Among these, the language *Javanese* [58] supplies primitives for declaring a context and its properties in a logical manner through a temporal logic. In *Javanese* the context represents properties of the system that are "activated by an action and held active until another action that deactivates it occurs". This is similar to our vision where the system running an application is part of the context and where a fact inserted into context holds until explicitly retracted. Also *Subjective-C* [46] is equipped with a domain-specific language for specifying the contents of what is called a *set of contexts*. A context of *Subjective-C* is just a single property holding in the working environment of an application, behaving much like our facts. Similarly, a context is activated when particular circumstances occur in the environment. Furthermore, *Subjective-C* proposes constructs for specifying relationships and constraints over contexts, e.g. inclusion and conflict. This approach is very similar to ours, and Datalog can also express these kinds of relations through logical rules.

Note that our approach loosely integrates a functional and a logical language differently from the tight integration offered by the standard functional logic programming (see, e.g. Bellia et al. [15] and Antoy and Hanus [2]). In this paper, the logical and the functional part are kept apart, and are only connected by the dispatching mechanism; our extended ML and Datalog (and the context) work in a sort of master/slave fashion.

In the current setting, the context is only updated by the applications, either by `tell/retract` or by invoking functions provided by the system API. More generally, the context can evolve independently of the applications, emitting events to signal the changes [3], [8] — implicitly representing the presence of many different applications sharing the same context. The major extensions to ML_{CoDa} for supporting these aspects include at least the ability of handling the concurrency between the context evolution and the running application, as well as primitives for reacting and adapting to events. The definition of an enhanced dynamic semantics requires minor modifications, also for managing the non-determinism due to the unknown order in which events show up. However, non-determinism may cause the history expressions computed at compile time to explode in size, making our analysis less effective. Further investigation is needed to devise techniques for reducing the search space, while maintaining the required expressivity.

As already mentioned, the present version of our dispatching mechanism has a fixed evaluation order, and the first case with a goal satisfied is selected, regardless if more cases can be taken. A more flexible approach would require Datalog to return a list of weights for all the expressions whose goals hold. The dispatching mechanism would be then parametrised on an adaptation policy that determines the best case. Also, Datalog rules *could* be defined to assign weights dynamically, e.g. using Bayesian learning techniques [9].

History expressions adapt the history effects by Skalka and Smith [84] and Skalka et al. [83] making them an intermediate verification language for analysing different program properties. They have been used for checking secure web service compositions [13], [12], [33] and resource usages [14], both within functional languages and in Java [11]. The approach taken here is typical of static program analysis [70]. A different, methodological one consists of writing programs at different level of abstractions, to be gradually refined, and expressed through specification languages. Then properties are proved at the suitable level. A typical example are the greybox specifications by Büchi and Weck [23], that have been used for proving properties of object oriented languages [82] and of web services [76].

Future work will address security issues, that are particularly relevant when applications move along different working environments. Modern programming languages include explicit primitives to specify and enforce security policies, that are not fully adequate for adaptive software. This is because they are mainly *defensive* and based on a threat model that is known a priori and does not change over time. These assumptions hold no longer when taking

context-awareness into account. Combining security and context-awareness requires to address two somehow conflicting aspects. On the one side, security requirements may reduce the adaptivity of software. On the other side, new highly dynamic security mechanisms are needed to scale up to adaptive software [95], [25]. A first investigation on this issues is in Bodei et al. [17], where ML_{CoDa} is extended to enforce security policies on the context modifications. An extension of our static analysis can detect potential violation of the required security policies, and also drive program instrumentation to prevent these to occur. This is a first example of a more sophisticated usage of evolution graphs.

We also plan to handle quantitative aspects of applications and contexts, by defining suitable extensions of Datalog and of History Expressions, see e.g. Degano et al. [43]. Also a more logically oriented verification technique is worth studying, in the style of Skalka et al. [83] and Bartoletti et al. [14]. Finally, it is worth investigating the integration of our static verification mechanism with a dynamic one, like those proposed by Calinescu et al. [24] and by Cardozo et al. [27].

ACKNOWLEDGMENT

The authors are deeply indebted with the anonymous reviewers for their invaluable comments and suggestions that greatly helped us to improve the presentation of our work.

REFERENCES

[1] Peter Alvaro, WilliamR. Marczak, Neil Conway, JosephM. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281. Springer Berlin Heidelberg, 2011.

[2] Sergio Antoy and Michael Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, April 2010.

[3] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. COP '11, pages 1:1–1:7, New York, NY, USA, 2011. ACM.

[4] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Unifying multiple layer activation mechanisms using one event sequence. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[5] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, USA, 2009. ACM.

[6] Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. Declarative layer composition with the JCop programming language. *Journal of Object Technology*, 12(2):4:1–37, June 2013.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[8] Engineer Bainomugisha. *Reactive method dispatch for Context-Oriented Programming*. PhD thesis, Comp. Sci. Dept., Vrije Universiteit Brussel, 2012.

[9] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[10] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: issue and challenges. *Computer*, 39(10):36–43, Oct 2006.

[11] Massimo Bartoletti, Gabriele Costa, Pierpaolo Degano, Fabio Martinelli, and Roberto Zunino. Securing Java with local policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[12] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Planning and verifying service composition. *Journal of Computer Security*, 17(5):799–837, 2009. abridged version in Proc. of CSFW 2005, IEEE Press, 211-223.

[13] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Semantics-based design for secure web services. *IEEE Trans. Software Eng.*, 34(1):33–49, 2008.

[14] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Local policies for resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009.

[15] Marco Bellia, Pierpaolo Degano, and Giorgio Levi. The call by name semantics of a clause language with functions. In K. L. Clatk and S.-A. Tärnlund, editors, *Logic Programming*, volume 16 of *APIC Studies in Data Processing*, pages 281–295. Academic Press, London, 1982.

[16] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 83–92, New York, NY, USA, 2004. ACM.

[17] Chiara Bodei, Pierpaolo Degano, Letterio Galletta, and Francesco Salvatori. Linguistic Mechanisms for Context-aware Security. In Gabriel Ciobanu and Dominique Méry, editors, *11th International Colloquium on Theoretical Aspects of Computing, ICTAC 2014*, volume 8687 of *Lecture Notes in Computer Science*. Springer, 2014.

[18] Rafael H. Bordini, Mehdi Dastani, Jrgen Dix, and Amal El Fallah Seghrouchni. *Multi-Agent Programming: Languages, Tools and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[19] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and IgorV. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer Berlin Heidelberg, 1993.

[20] Don Box and Chris Sells. *Essential .NET: The Common Language Runtime*. Microsoft .NET Development Series. Addison Wesley, 2002.

[21] Greg Brown, Betty H. C. Cheng, Heather Goldsby, and Ji Zhang. Goal-oriented specification of adaptation requirements engineering in adaptive systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 23–29, New York, NY, USA, 2006. ACM.

[22] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. A conceptual framework for adaptation. In Juan Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 240–254. Springer, 2012.

[23] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much, 1999. Turku Centre for Computer Science, Technical Report No 297.

[24] Radu Calinescu, Carlo Ghezzi, Marta Z. Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.

[25] Roy Campbell, Jalal Al-Muhtadi, Prasad Naldurg, Geetanjali Sampemane, and M. Mickunas. Towards security and privacy for pervasive computing. In Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *Software Security — Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2003.

[26] Andrea Canciani, Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A context-oriented extension of F#. In *FOCLASA 2015*, volume to appear of *EPTCS*, 2015.

[27] Nicolás Cardozo, Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. Run-time validation of behavioral adaptations. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[28] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, Jorge Vallejos, and Theo D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58(0):71 – 94, 2015.

[29] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, March 1989.

[30] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03):197–207, 9 2003.

[31] Dave Clarke, Pascal Costanza, and Éric Tanter. How should context-escaping closures proceed? In *International Workshop on*

Context-Oriented Programming, COP '09, pages 1:1–1:6, New York, NY, USA, 2009. ACM.

[32] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, COP '09, pages 10:1–10:6, New York, NY, USA, 2009. ACM.

[33] Gabriele Costa, Pierpaolo Degano, and Fabio Martinelli. Modular plans for secure service composition. *Journal of Computer Security*, 20(1):81–117, 2012.

[34] Pascal Costanza. Language constructs for context-oriented programming. In *In Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2005.

[35] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[36] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[37] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM.

[38] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, FrankS. Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer Berlin Heidelberg, 2013.

[39] Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A two-step type and effect inference for a context-aware language. submitted for publication, availabe at http://www.di.unipi.it/~galletta/Inference.pdf.

[40] Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A Two-Phase Static Analysis for Reliable Adaptation. In Dimitra Giannakopoulou and Grenoble Gwen Salaün, editors, *12th International Conference on Software Engineering and Formal Methods, SEFM 2014*, volume 8702 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2014.

[41] Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A two-component language for cop. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, pages 6:1–6:7, New York, NY, USA, 2014. ACM.

[42] Pierpaolo Degano, Gian Luigi Ferrari, Letterio Galletta, and Gianluca Mezzetti. Types for coordinating secure behavioural variations. In Marjan Sirjani, editor, *Coordination Models and Languages - 14th International Conference, COORDINATION 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 2012.

[43] Pierpaolo Degano, Gian-Luigi Ferrari, and Gianluca Mezzetti. On quantitative security policies. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 6873 of *Lecture Notes in Computer Science*, pages 23–39. Springer Berlin Heidelberg, 2011.

[44] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In Boicho Kokinov, DanielC. Richardson, ThomasR. Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context*, volume 4635 of *Lecture Notes in Computer Science*, pages 178–191. Springer Berlin Heidelberg, 2007.

[45] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In Rogério Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 61–89. Springer, 2003.

[46] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 246–265. Springer Berlin Heidelberg, 2011.

[47] T. Gu, X.H. Wang, H.K. Pung, and D.Q. Zhang. An ontology-based context model in intelligent environments. In *Proceedings of communication networks and distributed systems modeling and simulation conference*, volume 2004, pages 270–275, 2004.

[48] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 306–317, New York, NY, USA, 1994. ACM.

[49] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008.

[50] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: a minimal core calculus for context-oriented programming. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, pages 19–23. ACM, 2011.

[51] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing, Degrees, Models, and Applications. *ACM Comput. Surv.*, 40(3):1–28, August 2008.

[52] IBM. An architectural blueprint for autonomic computing. Technical report, June 2006.

[53] Atsushi Igarashi, Robert Hirschfeld, and Hidehiko Masuhara. A type system for dynamic layer composition. In *FOOL 2012*, page 13, 2012.

[54] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[55] Hiroaki Inoue, Atsushi Igarashi, Malte Appeltauer, and Robert Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. COP'14, pages 7:1–7:6, New York, USA, 2014. ACM.

[56] Tetsuo Kamina, Tomoyuki Aotani, and Atsushi Igarashi. On-demand layer activation for type-safe deactivation. COP'14, pages 4:1–4:7, New York, NY, USA, 2014. ACM.

[57] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A core calculus of composite layers. In *Proceedings of the 12th Workshop on Foundations of Aspect-oriented Languages*, FOAL '13, pages 7–12, New York, NY, USA, 2013. ACM.

[58] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. A unified context activation mechanism. In *Proceedings of the 5th International Workshop on Context-Oriented Programming*, COP'13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[59] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Tetsuo Tamai. Context-oriented software engineering: A modularity vision. MODULARITY '14, pages 85–98, New York, NY, USA, 2014. ACM.

[60] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[61] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[62] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and WilliamG. Griswold. An Overview of AspectJ. In JørgenLindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin Heidelberg, 2001.

[63] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240 – 266, 2006.

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

[65] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, October 1996.

[66] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with circles, triangles and rectangles. In *In XML Conference and Exposition*, 2003.

[67] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 90–99, New York, NY, USA, 2003. ACM.

[68] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295 – 307, 1984.

[69] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer Berlin Heidelberg.

[70] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1st ed. 1999. corr. 2nd printing, 1999 edition, 2005.

[71] Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In TorbenÆ. Mogensen, David A. Schmidt, and I.Hal Sudborough, editors, *The Essence*

of Computation, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Berlin Heidelberg, 2002.

[72] Giorgio Orsi and Letizia Tanca. Context modelling and context-aware querying. In Oege Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 225–244. Springer, 2011.

[73] Mike P. Papazoglou and Dimitrios Georgakopoulos. Introduction: Service-oriented computing. *Commun. ACM*, 46(10):24–28, October 2003.

[74] Christian Peper and Daniel Schneider. Component engineering for adaptive ad-hoc systems. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 49–56, New York, NY, USA, 2008. ACM.

[75] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[76] Hridesh Rajan, Jia Tao, Steve M. Shaner, and Gary T. Leavens. Tisa: A language design and modular verification technique for temporal policies in web services. In *ESOP 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2009.

[77] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51, Washington, DC, USA, 2009. IEEE Computer Society.

[78] Eva Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, January 2004.

[79] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[80] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR*, abs/1105.0069, 2011.

[81] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 8(2):7:1–7:29, July 2013.

[82] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Montreal, Quebec, Canada*, pages 351–368. ACM, 2007.

[83] Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, 18(2):179–249, 2008.

[84] Christian Skalka and Scott F. Smith. History effects and verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128. Springer, 2004.

[85] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. CRPIT '02, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[86] Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 45–53, New York, NY, USA, 1995. ACM.

[87] Éric Tanter. Contextual values. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 3:1–3:10, New York, NY, USA, 2008. ACM.

[88] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th International Conference on Aspect-oriented Software Development*, AOSD '08, pages 168–179, New York, NY, USA, 2008. ACM.

[89] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions. In Benoît Baudry and Eric Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 66–81. Springer Berlin Heidelberg, 2010.

[90] Bart van Wissen, Nicholas Palmer, Roelof Kemp, Thilo Kielmann, and Henri Bal. ContextDroid: an expression-based context framework for Android. In *Proceedings of PhoneSense 2010*, 2010.

[91] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.

[92] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. *SIGPLAN Not.*, 38(9):127–139, August 2003.

[93] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, September 2004.

[94] X.H. Wang, D.Q. Zhang, T. Gu, and H.K. Pung. Ontology based context modeling and reasoning using OWL. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. Ieee, 2004.

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

Pierpaolo Degano has been a full professor of computer science since 1990 and he has been with the Department of Computer Science at the University of Pisa since 1993. He is member of the Editorial Board of Theoretical Computer Science, of two sub-series in the Monographs & Texts in Theoretical Computer Science and of Mondo Digitale; he served as editor of many special issues of scientific journals and of conference proceedings. Pierpaolo Degano is a co-founder, and since 2012 chairman, of the IFIP TC1WG1.7 on Theoretical Foundations of Security Analysis and Design, also he a member of the board of directors of the Microsoft Research-University of Trento Center for Computational and Systems Biology. His research interests are broadly in Programming Languages, and include security of concurrent and mobile systems, systems biology, semantics and concurrency, methods and tools for program verification and evaluation, and programming tools.

Gian Luigi Ferrari received the PhD degree in computer science from the University of Pisa, where he is Full Professor at the Department of Computer Science since 2011. His present research interests include formal methods for software engineering, program analysis, security for service oriented applications and programming languages design for pervasive software systems.

Letterio Galletta received the PhD degree in Computer Science from the University of Pisa in 2014. His research activity ranges mainly from programming languages to static analyses for checking correctness and safety of programs. His current research interests include programming languages for adaptive software, type and effect systems and language-based security.

APPENDIX

ML$_{CoDA}$ SEMANTICS

Below we list all the rules of the ML$_{CoDa}$ SOS semantics.

(PAR)
$$\frac{\rho(\hat{x}) = Va \qquad dsp(C, Va) = (e, \{\overrightarrow{c}/\overrightarrow{y}\})}{\rho \vdash C, \hat{x} \to C, e\{\overrightarrow{c}/\overrightarrow{y}\}}$$

(TELL1)
$$\frac{\rho \vdash C, e \to C', e'}{\rho \vdash C, tell(e) \to C', tell(e')}$$

(TELL2)
$$\frac{}{\rho \vdash C, tell(F) \to C \cup \{F\}, ()}$$

(RETRACT1)
$$\frac{\rho \vdash C, e \to C', e'}{\rho \vdash C, retract(e) \to C', retract(e')}$$

(RETRACT2)
$$\frac{}{\rho \vdash C, retract(F) \to C \backslash \{F\}, ()}$$

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

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

(APPEND1)
$$\frac{\rho \vdash C, e_1 \to C', e_1'}{\rho \vdash C, e_1 \cup e_2 \to C', e_1' \cup e_2}$$

(APPEND2)
$$\frac{\rho \vdash C, e_2 \to C', e_2'}{\rho \vdash C, (x)\{Va_1\} \cup e_2 \to C', (x)\{Va_1\} \cup e_2'}$$

(APPEND3)
$$\frac{z \text{ fresh}}{\rho \vdash C, (x)\{Va_1\} \cup (y)\{Va_2\} \to C, (z)\{Va_1\{z/x\}, Va_2\{z/y\}\}}$$

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

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

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

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

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

(APP1)
$$\frac{\rho \vdash C, e_1 \to C', e_1'}{\rho \vdash C, e_1\, e_2 \to C', e_1'\, e_2}$$

(APP2)
$$\frac{\rho \vdash C, e_2 \to C', e_2'}{\rho \vdash C, (\lambda_f x.e)\, e_2 \to C', (\lambda_f x.e)\, e_2'}$$

(APP3)
$$\frac{}{\rho \vdash C, (\lambda_f x.e)\, v \to C, e\{v/x, (\lambda_f x.e)/f\}}$$

(VAAPP1)
$$\frac{\rho \vdash C, e_1 \to C', e_1'}{\rho \vdash C, \#(e_1, e_2) \to C', \#(e_1', e_2)}$$

(VAAPP2)
$$\frac{\rho \vdash C, e_2 \to C', e_2'}{\rho \vdash C, \#((x)\{Va\}, e_2) \to C', \#((x)\{Va\}, e_2')}$$

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

TYPE AND EFFECT SYSTEM

Below we list all the typing rules of the type and effect system for ML_{CoDa}.

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

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

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

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

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

(TFACT)
$$\frac{}{\Gamma;\, K \vdash F \,:\, fact_{\{F\}} \triangleright \epsilon}$$

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

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

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

(TTELL)
$$\frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H}{\Gamma;\, K \vdash tell(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} tell\, F \right)}$$

(TRETRACT)
$$\frac{\Gamma;\, K \vdash e \,:\, fact_\phi \triangleright H}{\Gamma;\, K \vdash retract(e) \,:\, unit \triangleright H \cdot \left(\sum_{F \in \phi} retract\, F \right)}$$

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

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

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

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

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

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

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

(TDLET)
$$\frac{\Gamma, \overrightarrow{y} : \overrightarrow{\tau};\, K \vdash e_1 \,:\, \tau_1 \triangleright H_1 \qquad \Gamma;\, K, (\hat{x},\, \tau_1,\, \Delta') \vdash e_2 \,:\, \tau_2 \triangleright H_2}{\Gamma;\, K \vdash dlet\, \hat{x} \,=\, e_1\, when\, G\, in\, e_2 \,:\, \tau_2 \triangleright H_2}$$

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

Properties of the Type and Effect System

Below we prove Theorem 6.1, Theorem 6.2 and Corollary 6.3. We start giving some lemmas and definitions useful for the proofs formal development.

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

$$
\begin{aligned}
&c\{e'/x\} = c\\
&F\{e'/x\} = F\\
&(\lambda_f x'.e)\{e'/x\} = \lambda_f x'.e\{e'/x\}\\
&\qquad\qquad \text{if } f \neq x \wedge x' \neq x \wedge f, x' \notin FV(e')\\
&(x')\{G_1.e_1, \ldots, G_n.e_n\}\{e'/x\} =\\
&\qquad (x')\{G_1.e_1\{e'/x\}, \ldots, G_n.e_n\{e'/x\}\}\\
&\qquad\qquad \text{if } x \neq x' \wedge x \in \bigcup_{i\in\{1,\ldots,n\}} FV(G_i) \wedge\\
&\qquad\qquad \left(\{x'\} \cup \bigcup_{i\in\{1,\ldots,n\}} FV(G_i)\right) \cap FV(e') = \emptyset\\
&x\{e'/x\} = e'\\
&x'\{e'/x\} = x' \qquad\qquad \text{if } x \neq x'\\
&(e_1\, e_2)\{e'/x\} = e_1\{e'/x\}\, e_2\{e'/x\}\\
&(e_1\, op\, e_2)\{e'/x\} = e_1\{e'/x\}\, op\, e_2\{e'/x\}\\
&(if\, e_1\, then\, e_2\, else\, e_3)\{e'/x\} =\\
&\qquad if\, e_1\{e'/x\}\, then\, e_2\{e'/x\}\, else\, e_3\{e'/x\}\\
&(tell(e))\{e'/x\} = tell(e\{e'/x\})\\
&(retract(e))\{e'/x\} = retract(e\{e'/x\})\\
&(e_1 \cup e_2)\{e'/x\} = e_1\{e'/x\} \cup e_2\{e'/x\}\\
&\#(e_1, e_2)\{e'/x\} = \#(e_1\{e'/x\}, e_2\{e'/x\})\\
&(let\, x' = e_1\, in\, e_2)\{e'/x\} = let\, x' = e_1\{e'/x\}\, in\, e_2\{e'/x\}\\
&\qquad\qquad \text{if } x \neq x' \wedge x' \in FV(e')\\
&(dlet\, \hat{x} = e_1\, when\, G\, in\, e_2)\{e'/x\} =\\
&\qquad dlet\, \hat{x} = e_1\{e'/x\}\, when\, G\, in\, e_2\{e'/x\}\\
&\qquad\qquad \text{if } x \notin FV(G) \wedge FV(G) \cap FV(e') = \emptyset.
\end{aligned}
$$

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

Proof. The thesis follows from Definition 6.1 and that of $K \sqsubseteq K'$. □

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

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

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

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

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

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

Proof. The thesis follows by using the definition 6.1 and that of $K \sqsubseteq K'$. □

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

Proof. Straightforward induction on typing derivations. □

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

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

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

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

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

Proof.

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

□

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

1) *$\Gamma; K \vdash v : \tau_c \triangleright H$ then $v = c$*
2) *$\Gamma; K \vdash v : \tau_1 \xrightarrow{K'|H'} \tau_2 \triangleright H$ then $v = \lambda_f x.e$*
3) *$\Gamma; K \vdash v : \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright H$ then $v = (x)\{Va\}$*
4) *$\Gamma; K \vdash v : fact_{\{F_1,\ldots,F_m\}} \triangleright H$ then $v \in \{F_1, \ldots, F_m\}$*

Proof.

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

□

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

1) *If $\Gamma; K \vdash \lambda_f x.e : \tau_1 \xrightarrow{K'|H} \tau_2 \triangleright H'$ and $K' \sqsubseteq K$ then $\Gamma, x : \tau_1, f : \tau_1 \xrightarrow{K'|H} \tau_2; K \vdash e : \tau_2 \triangleright H$*
2) *If $\Gamma; K \vdash (x)\{G_1.e_1, \ldots, G_n.e_n\} : \tau_1 \xRightarrow{K'|\Delta} \tau_2 \triangleright H'$ and $K' \sqsubseteq K$ and $\Delta = \bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i$ then*

$\forall i \in \{1,\ldots,n\}\ \Gamma, x : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K \vdash e_i : \tau_2 \triangleright H_i$ *where* $\overrightarrow{y_i} : \overrightarrow{\tau_i} = \gamma(G_i)$

Proof.

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

□

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

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

- rule (TTELL)
  By the premise of the rule we know that $\Gamma, x : \tau'; K \vdash e : fact_\phi \triangleright H'$ holds. By using the induction hypothesis we can claim that $\Gamma; K \vdash tell(e\{v/x\}) : \tau \triangleright H$ and by Definition A.1 we can conclude that $\Gamma; K \vdash (tell(e))\{v/x\} : \tau \triangleright H$.
- rule (TRETRACT)
  Similar to the case (TTELL)
- rule (TAPPEND)
  By the premise of the rule we know that $\Gamma, x : \tau'; K \vdash e_i : \tau_1 \xrightarrow{K'|\Delta_i} \tau_2 \triangleright H_i$ for $i \in \{1,2\}$ holds. By the inductive hypothesis we can claim that $\Gamma; K \vdash e_1\{v/x\} \cup e_2\{v/x\} : \tau \triangleright H$ holds. By Definition A.1 we conclude $\Gamma; K \vdash (e_1 \cup e_2)\{v/x\} : \tau \triangleright H$.
- rule (TVAPP)
  By the premise of the rule we know that $\Gamma, x : \tau'; K \vdash e_1 : \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright H_1$ and $\Gamma, x : \tau'; K \vdash e_2 : \tau_1 \triangleright H_2$ and $K' \sqsubseteq K$. By using the induction hypothesis we can claim that $\Gamma; K \vdash \#(e_1\{v/x\},\, e_2\{v/x\}) : \tau \triangleright H$ holds and by Definition 6.1 we can conclude that $\Gamma; K \vdash \#(e_1,\, e_2)\{v/x\} : \tau \triangleright H$.
- rule (TVARIATION)
  By the premise of the rule (TVARIATION) we know that $\forall i \in \{1,\ldots,n\}\ \Gamma, x : \tau', x' : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}; K' \vdash e_i : \tau_2 \triangleright H_i$ where $\overrightarrow{y_i} : \overrightarrow{\tau_i} = \gamma(G_i)$, $\Delta = \bigotimes_{i\in\{1,\ldots,n\}} ask\, G_i.H_i$. By Lemma A.4 $\forall i \in \{1,\ldots,n\}\ \Gamma, x' : \tau_1, \overrightarrow{y_i} : \overrightarrow{\tau_i}, x : \tau'; K' \vdash e_i : \tau_2 \triangleright H_i$. By using the induction hypothesis and the rule (TVARIATION) we can claim that $\Gamma; K \vdash (x')\{G_1.e_1\{v/x\}, \ldots, G_n.e_n\{v/x\}\} : \tau \triangleright H$ and by Definition A.1 we conclude $\Gamma; K \vdash (x')\{G_1.e_1, \ldots, G_n.e_n\}\{v/x\} : \tau \triangleright H$.
- rule (TDLET)
  By the precondition of the rule (TDLET) we know that $\Gamma, x : \tau', \overrightarrow{y} : \overrightarrow{\tau}; K \vdash e_1 : \tau_1 \triangleright H_1$ and $\Gamma, x : \tau'; K, (\hat{x}, \tau_1, \Delta) \vdash e_2 : \tau \triangleright H_2$ with $\overrightarrow{y} : \overrightarrow{\tau} = \gamma(G)$. By Lemma A.4 $\Gamma, \overrightarrow{y} : \overrightarrow{\tau}, x : \tau'; K \vdash e_1 : \tau_1 \triangleright H_1$. By using the induction hypothesis we can claim that $\Gamma; K \vdash dlet\, \hat{x} = e_1\{v/x\}\, when\, G\, in\, e_2\{v/x\} : \tau \triangleright H_2$ and by Definition A.1 $\Gamma; K \vdash (dlet\, \hat{x} = e_1\, when\, G\, in\, e_2)\{v/x\} : \tau \triangleright H_2$.
- rule (TCONST), (TFACT), (TDVAR) Since $e\{v/x\} = e$ by Definition A.1 the result $\Gamma; K \vdash e : \tau \triangleright H$ is immediate, when $e = c$, $e = F$ and $e = \hat{x}$.
- The other cases are standard.

□

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

Proof. Similar to that of Lemma A.9 □

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

Proof. In the typing derivation for the judgement $\Gamma; K \vdash v : \tau \triangleright H$ there is a subderivation with conclusion $\Gamma; K \vdash v : \tau' \triangleright \epsilon$ for some τ'. This conclusion is obtained by applying one of typing rules for values. Since v is a value we can obtain $\Gamma; K \vdash v : \tau \triangleright H$ from this conclusion by applying only the rule (Tsub) to enlarge the type and the effect. So we can make a new derivation that simulates the first one but where we enlarge only the type but not the effect. In this way we constructed a derivation for the judgement $\Gamma; K \vdash v : \tau \triangleright \epsilon$. □

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

Proof. In the typing derivation for the judgement $\Gamma; K \vdash v : \tau \triangleright H$ there is a subderivation with conclusion $\Gamma; K \vdash v : \tau' \triangleright \epsilon$ for some τ'. This conclusion is obtained by applying one of typing rules for values. Since v is a value we can obtain $\Gamma; K \vdash v : \tau \triangleright H$ from this conclusion by applying only the rule (Tsub) to enlarge the type and the effect. Then the thesis follows by the definition of $\sqsubseteq$. □

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

Proof. By induction of the depth typing derivation. □

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

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

Proof. Item (1) is immediate by applying the rule for summation. Item (ii) follows by induction on the length of the computation $C, H \to^* C', H'$, applying the rule for $\cdot$. □

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

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

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

Property A.17. *If* $H \sqsubseteq H'$ *then* $H \cdot H'' \sqsubseteq H' \cdot H''$.

Theorem 6.1 (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 $H \sqsubseteq H'_s$.

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

In the proof we implicitly use the fact that $H \sqsubseteq H$ for each H (except for the case TSUB).

- rule (TVARIATION) or (TCONST) or (TFACT) or (TABS) or (TVAR)
  In this case we know that e_s is a value (or a variable in the case (TVAR)), then for no e'_s it holds $\rho \vdash C,\ e_s \to C',\ e'_s$, so the theorem holds vacuously.
- rule (TTELL)
  We know that $e_s = tell(e')$ for some e' and also by the (TTELL) premise that $\Gamma; K \vdash e'\ :\ fact_\phi \triangleright H$ holds and $H_s = H \cdot \sum_{F\in\phi} tell\ F$. We have only two rules by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.
  - rule (TELL1)
    We know that e' is an expression and $e'_s = tell(e'')$ and $\rho \vdash C,\ e' \to C',\ e''$ and there is in our derivation a subderivation with conclusion $\Gamma; K \vdash e'\ :\ fact_\phi \triangleright H$. By the induction hypothesis $\Gamma; K \vdash e''\ :\ fact_\phi \triangleright H''$ and that $C,\ H \to^\star C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H''$. By using the rule (TTELL) we can conclude that $\Gamma; K \vdash e'_s\ :\ unit \triangleright H'_s$ and $H'_s = H'' \cdot \sum_{F\in\phi} tell\ F$. Lemma A.14 now suffices for establishing that $C, H \cdot \sum_{F\in\phi} tell\ F\ \to^\star\ C', \overline{H} \cdot \sum_{F\in\phi} tell\ F$ and by Property A.17 $\overline{H} \cdot \sum_{F\in\phi} tell\ F \sqsubseteq H'' \cdot \sum_{F\in\phi} tell\ F$.
  - rule (TELL2)
    We now that $e' = F$, $e'_s = ()$ and $C' = C \cup \{F\}$. We have to prove that $\Gamma; K \vdash e'_s\ :\ unit \triangleright H'_s$, but from the rule (TCONST) we know that this holds with $H'_s = \epsilon$. It remains to show that $C, H_s\ \to^*\ C', H'_s$. From Lemma A.12 we know $H_s = \epsilon + \Sigma_{i=1}^n H_i$. Then, $C, (\epsilon + \Sigma_{i=1}^n H_i) \cdot \Sigma_{F\in\phi} tell\ F\ \to\ C, \epsilon \cdot \Sigma_{F\in\phi} tell\ F\ \to\ \Sigma_{F\in\phi} tell\ F\ \to\ C, tell\ F\ \to\ C \cup \{F\}, \epsilon = C', H'_s$.
- rule (TRETRACT)
  Similar to (TTELL) rule ($retract$ substitutes $tell$)
- rule (TAPPEND)
  We know $e_s = e_1 \cup e_2$ and $\tau = \tau_1 \xrightarrow{K'|\Delta_1\otimes\Delta_2} \tau_2$ and $H_s = H_1 \cdot H_2$, and also by the premise of (TAPPEND) that $\Gamma; K \vdash e_1\ :\ \tau_1 \xrightarrow{K'|\Delta_1} \tau_2 \triangleright H_1$ and $\Gamma; K \vdash e_2\ :\ \tau_1 \xrightarrow{K'|\Delta_2} \tau_2 \triangleright H_2$ hold. There are three rules only by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.
  - rule (APPEND1)
    We know that e_1 and e_2 are not values and $e'_s = e'_1 \cup e_2$. By applying the induction hypothesis $\Gamma; K \vdash e'_1\ :\ \tau_1 \xrightarrow{K'|\Delta_1} \tau_2 \triangleright H'_1$ with $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$. By applying the (TAPPEND) rule we can conclude that $\Gamma; K \vdash e'_1 \cup e_2\ :\ \tau_1 \xrightarrow{K'|\Delta_1\otimes\Delta_2} \tau_2 \triangleright H'_1 \cdot H_2$. The thesis follows by applying Lemma A.14 and Property A.17.
  - rule (APPEND2)
    We know that $e'_s = (x)\{Va_1\} \cup e'_2$. By applying the induction hypothesis $\Gamma; K \vdash e'_2\ :\ \tau_1 \xrightarrow{K'|\Delta_2} \tau_2 \triangleright H'_2$ with $C,\ H_2 \to^*\ C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H'_2$. By the rule (TVARIATION) we know that $\Gamma; K\ \vdash\ (x)\{Va_1\}\ :\ \tau_1 \xrightarrow{K'|\Delta_1} \tau_2 \triangleright \epsilon$ and by applying the rule (TAPPEND) we can claim that $\Gamma; K\ \vdash\ (x)\{Va_1\} \cup e'_2\ :\ \tau_1 \xrightarrow{K'|\Delta_1\otimes\Delta_2} \tau_2 \triangleright \epsilon \cdot H'_2 = H'_2 = H'_s$ by the Property A.15. By Lemma A.12 we know $H_1 = (\epsilon + \Sigma_{i=1}^n H_i)$, then $C, H_s = C, H_1 \cdot H_2 \to C, \epsilon \cdot H_2 \to C, H_2 \to^* C', \overline{H}$, proving the thesis since $\overline{H} \sqsubseteq H'_s$.
  - rule (APPEND3)
    We know that e_s is

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

    and that e'_s is

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

    By the premise of the rule (TVARIATION), we also know that $\forall i \in \{1, \ldots, n\}$ we have $\Gamma, x\ :\ \tau_1, \overrightarrow{y_i}\ :\ \overrightarrow{\tau_i}; K' \vdash e_i\ :\ \tau_2 \triangleright H_i$ and $\forall j \in \{1, \ldots, m\}$ we have $\Gamma, y\ :\ \tau_1, \overrightarrow{y_j}\ :\ \overrightarrow{\tau_j}; K' \vdash e'_j\ :\ \tau_2 \triangleright H_j$. By Lemma A.10 it holds that $\forall i \in \{1, \ldots, n\}\ \Gamma, z\ :\ \tau_1, \overrightarrow{y_i}\ :\ \overrightarrow{\tau_i}; K' \vdash e_i\{z/x\}\ :\ \tau_2 \triangleright H_i$ and $\forall j \in \{1, \ldots, m\}\ \Gamma, z\ :\ \tau_1, \overrightarrow{y_j}\ :\ \overrightarrow{\tau_j}; K' \vdash e'_j\{z/x\}\ :\ \tau_2 \triangleright H_j$. So by applying the rule (TVARIATION) for all judgements indexed by i and j we can conclude that $\Gamma; K \vdash e'_s\ :\ \tau_1 \xrightarrow{K'|\Delta_1\otimes\Delta_2} \tau_2 \triangleright \epsilon = H'_s$. By applying twice Lemma A.12 we have $H_j = (\epsilon + \Sigma_{i=1}^n H_i)$ for $j \in \{1, 2\}$. Then, the thesis follows because $C, H_s = C, H_1 \cdot H_2 \to^* C, H_2 \to C, \epsilon$.
- rule (TVAPP)
  We know that $\Gamma; K \vdash e_1\ :\ \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright H_1$, $\Gamma; K \vdash e_2\ :\ \tau_1 \triangleright H_2$ and $K' \sqsubseteq K$ hold by (TVAPP) premises. There are three rules only by which $\rho \vdash C,\ e_s \to C',\ e'_s$ can be derived.
  - rule (VAPP1)
    We know that $e'_s = \#(e'_1,\ e_2)$. By the induction hypothesis $\Gamma; K \vdash e'_1\ :\ \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright H'_1$ with $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$. By (TVAPP) rule we have $\Gamma; K\ \vdash\ e'_s\ :\ \tau_2 \triangleright H'_1 \cdot H_2 \cdot \Delta$. By Lemma A.14 we can conclude $C, H_1 \cdot H_2 \cdot \Delta \to^* C', \overline{H} \cdot H_2 \cdot \Delta$ and the thesis follows by Property A.17.
  - rule (VAPP2)
    We know that $e'_s = \#((x)\{Va\},\ e'_2)$. By using Lemma A.11 we have $\Gamma; K\ \vdash\ (x)\{Va\}\ :\ \tau_1 \xrightarrow{K'|\Delta} \tau_2 \triangleright \epsilon$ and by the induction hypothesis $\Gamma; K \vdash e'_2\ :\ \tau_1 \triangleright H'_2$ with $C,\ H_2 \to^*\ C',\ \overline{H}$ for some $\overline{H} \sqsubseteq H'_2$. By (TVAPP) and Property A.15 $\Gamma; K \vdash e'_s\ :\ \tau_2 \triangleright \epsilon \cdot H'_2 \cdot \Delta = H'_2 \cdot \Delta$ holds. By Lemma A.12 we have $H_1 = (\epsilon + \Sigma_{i=1}^n H_i)$, then

$C, H_1 \cdot H_2 \cdot \Delta \to C, \epsilon \cdot H_2 \cdot \Delta \to C, H_2 \cdot \Delta$. By Lemma A.14 we have $C, H_2 \cdot \Delta \to^* C', \overline{H} \cdot \Delta$ and Property A.17 proves the thesis.

- rule (VAPP3)
  We know that $e_s = \#((x)\{Va\}, v)$ where $Va = G_1.e_1, \ldots, G_n.e_n$, $e'_s = e_j\{v/x, \overrightarrow{c}/\overrightarrow{y}\}$ for $j \in \{1, \ldots, n\}$ and $\rho \vdash C, e_s \to C, e'_s$. From our hypothesis and from Lemma A.8(2) we have that for all $i \in \{1, \ldots, n\}$ it holds $\Gamma, x : \tau_1, \overrightarrow{y_i} : \overrightarrow{t_i}; K \vdash e_i : \tau_2 \triangleright H_i$. By Lemma A.11 we also know that $\Gamma; K \vdash v : \tau_1 \triangleright \epsilon$. So by Lemma A.9 we have that for $i \in \{1, \ldots, n\}$ $\Gamma; K \vdash e_i\{v/x, \overrightarrow{\tau}/\overrightarrow{y}\} : \tau \triangleright H_i$. By Lemma A.12 we have $H_j = \epsilon$ for $j \in \{1, 2\}$, then $C, H_1 \cdot H_2 \cdot \Delta \to C, \epsilon \cdot H_2 \cdot \Delta \to C, H_2 \cdot \Delta \to C, \epsilon \cdot \Delta \to C, \Delta$. The thesis follows by using Lemma A.14.

- rule (TDLET)
  If the last rule in the derivation is (TDLET) we know that there is a subderivation with conclusions $\gamma(G) = \overrightarrow{y} : \overrightarrow{\tau}$ and $\Gamma, \overrightarrow{y} : \overrightarrow{\tau}; K \vdash e_1 : \tau_1 \triangleright H_1$ and $\Gamma; K_{\hat{x}}, (\hat{x}, \tau_1\, \Delta') \vdash e_2 : \tau \triangleright H$ and $\Delta' = askG.H_1$ when $\hat{x} \notin dom(K)$ or $\Delta' = askG.H_1 \otimes \Delta$ when $K(\hat{x}) = (\tau_1, \Delta)$. There are two rules by which $\rho \vdash C, e_s \to C', e'_s$ can be derived.
  - rule (DLET1)
    We know that $e'_s = dlet\, \hat{x} = e_1\, when\, G\, in\, e'_2$ and $\rho' \vdash C, e_2 \to C', e'_2$ with $\rho' = \rho[G.e_1, \rho(\hat{x})/\hat{x}]$. By Lemma A.1 $\Gamma \vdash \rho : K'$ with $K' = K_{\hat{x}}, (\hat{x}, \tau, \Delta')$ and by Lemma A.2 we know that $\Gamma \vdash \rho' : K'$. So by induction hypothesis $\Gamma; K' \vdash e'_2 : \tau \triangleright H'$ with $C, H \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'$. The judgement $\Gamma; K \vdash e'_s : \tau \triangleright H'$ follows by applying the rule (TDLET).
  - rule (DLET2)
    We know that $e'_s = v$ and $\rho \vdash C, e_s \to C, e'_s$. By hypothesis we know that $\Gamma; K_{\hat{x}}, (\hat{x}, \tau_1, \Delta') \vdash v : \tau \triangleright H$ and by the Lemma A.13 we have $\Gamma; K \vdash v : \tau \triangleright H$ and the thesis follows by choosing vacuously.

- rule (TDVAR)
  By the premise of rule (TDVAR) $K(\hat{x}) = (\tau, \Delta)$, where $\Delta = \bigotimes_{i \in \{1, \ldots, n\}} ask\, G_i.H_i \otimes fail$. We have to prove that if $\rho \vdash C, \hat{x} \to C', e$ then $\Gamma; K \vdash e : \tau \triangleright H'$. By the premises of the rule we know that $\rho(\hat{x}) = G_1.e_1, \ldots, G_n.e_n$ and that there exists a $j \in \{1, \ldots, n\}$ such that $e = e_j$. Since $\Gamma \vdash \rho : K$ we have that for all $i \in \{1, \ldots, n\}$ it holds $\Gamma, \overrightarrow{y_i} : \overrightarrow{\tau_i} \vdash e_i : \tau \triangleright H_i$ where $\gamma(G_i) = \overrightarrow{y_i} : \overrightarrow{\tau_i}$ and by Lemma A.9 we conclude that $\Gamma; K \vdash e_i\{\overrightarrow{t_i}/\overrightarrow{y_i}\} : \tau \triangleright H_i$ for all $i \in \{1, \ldots, n\}$. The thesis holds from Lemma A.14.

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

- rule (TLET)
  By the premise of rule (TLET) we know that $\Gamma; K \vdash e_1 : \tau_1 \triangleright H_1$ and $\Gamma, x : \tau_1; K \vdash e_2 : \tau_2 \triangleright H_2$ hold. There are only two rules by which $\rho \vdash C, e_s \to C', e'_s$ can be derived.
  - rule (LET1)
    We know that $e'_s = let\, x = e'_1\, in\, e_2$. By the induction hypothesis we have that $\Gamma; K \vdash e'_1 : \tau_1 \triangleright H'_1$ and $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$ and $\Gamma; K \vdash e'_s : \tau_2 \triangleright H'_1 \cdot H_2$. The thesis follows by Lemma A.14 and Property A.17.
  - rule (LET2)
    We know that $e'_s = e_2\{v/x\}$, $\rho \vdash C, e_s \to C, e'_s$, $\Gamma; K \vdash v : \tau_1 \triangleright H_1$ and $\Gamma, x : \tau_1; K \vdash e_2 : \tau_2 \triangleright H_2$. By Lemma A.11 $\Gamma; K \vdash v : \tau_1 \triangleright \epsilon$ and by Lemma A.9 $\Gamma; K \vdash e_2\{v/x\} : \tau_2 \triangleright H_2$. By Lemma A.12 we have $H_1 = (\epsilon + \sum_{i=1}^{n} H_i)$, so $C, H_1 \cdot H_2 \to C, \epsilon \cdot H_2 \to C, H_2$ proving the thesis.

- rule (TIF)
  By the premise of rule (TIF) we know that $\Gamma; K \vdash e_1 : bool \triangleright H_1$, $\Gamma; K \vdash e_2 : \tau \triangleright H_2$ and $\Gamma; K \vdash e_3 : \tau \triangleright H_3$ hold. There are three rules only by which $\rho \vdash C, e_s \to C', e'_s$ can be derived.
  - rule (IF1)
    We know that $e'_s = if\, e'_1\, then\, e_2\, else\, e_3$. By using the induction hypothesis we have that

$\Gamma; K \vdash e'_1 : bool \triangleright H'_1$ with $C, H_1 \to^* C', \overline{H}$ for some $\overline{H} \sqsubseteq H'_1$. So by rule (TIF) we conclude that $\Gamma; K \vdash e'_s : \tau \triangleright H'_1 \cdot (H_2 + H_3)$ and the thesis follows by Lemma A.14 and Property A.17.

- rule (IF2)
  We know that $e'_s = e_2$, $\rho \vdash C, e_s \to C, e'_s$ and $\Gamma; K \vdash e_2 : \tau \triangleright H_2$. By Lemma A.12 we know $H_1 = (\epsilon + \sum_{i=1}^{n} H_i)$, so the thesis is immediate because $C, H_1 \cdot (H_2 + H_3) \to C, \epsilon \cdot (H_2 + H_3) \to C, H_2$.
- rule (IF3)
  Similar to rule (IF2)

- rule (TSUB)
  By the premise of rule (TSUB) we know $\Gamma; K \vdash e_s : \tau' \triangleright H'$, $\tau' \leq \tau$ and $H_s = H' + \overline{H}$. Then by the induction hypothesis $\Gamma; K \vdash e'_s : \tau' \triangleright H'_1$, and $C, H' \to^* C', H'_2$ for some $H'_2 \sqsubseteq H'_1$. By applying the (TSUB) rule with $H'_s = H'_1 + \overline{H}$ we have $\Gamma; K \vdash e'_s : \tau' \triangleright H'_s$. The thesis follows because $C, H_s = C, H' + \overline{H} \to C, H' \to^* C', H'_2$ and because $H'_2 \sqsubseteq H'_1 \sqsubseteq H'_s$.

□

Proposition 6.3 (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 $H' \sqsubseteq H'_s$.*

Proof. An easy inductive reasoning on the length of the computation then suffices to prove the statement by using Theorem 6.1. □

Theorem 6.2 (Progress)**.** *Let e_s be a closed expression such that $\Gamma; K \vdash e_s : \tau \triangleright H_s$; and let ρ be a dynamic environment such that $dom(\rho)$ includes the set of parameters of e_s, and such that $\Gamma \vdash \rho : K$. If $\rho \vdash C, e_s \not\to \wedge C, H_s \not\to^+ C', fail$ then e_s is a value.*

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

- rule (TIF) $e_s = if\, e_1\, then\, e_2\, else\, e_3$
  If e_s is stuck, then it is only the case that e_1 is stuck. By induction hypothesis this can occur only when e_1 is a value. Since $\Gamma; K \vdash e_1 : bool \triangleright H_1$ by our hypothesis and $v = true$ or $v = false$ by Lemma A.7(1), either rule (IF2) or (IF3) applies, contradiction.
- rule (TLET) $e_s = let\, x = e_1\, in\, e_2$
  If e_s is stuck, then it is only the case that e_1 is stuck. By induction hypothesis this can occur only when e_1 is a value, hence (LET2) rule applies, contradiction.
- rule (TTELL) $e_s = tell(e)$
  If e_s is stuck, then it is only the case that e is stuck. By induction hypothesis this can occur only when e is a value v and by Lemma A.7(4) $v = F$, so the rule (TELL2) applies, contradiction.
- rule (TRETRACT) $e_s = retract(e)$
  Similar to the (TTELL) case.
- rule (TAPPEND) $e_s = e_1 \cup e_2$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(3) $e_1 = (x)\{Va\}$. If e_2 reduces, rule (VAPPEND2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value and Lemma A.7(3) $e_2 = (y)\{Va\}$, hence, rule (VAPPEND3) applies, contradiction.
- rule (TSUB)
  Straightforward by induction hypothesis
- rule (TAPP) $e_s = e_1\, e_2$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(2) $e_1 = \lambda_f x.e$. If e_2 reduces, rule (APP2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value, hence, rule (APP3) applies, contradiction.
- rule (TDVAR) $e_s = \hat{x}$
  If e_s is stuck we can have two cases only. The first case is that $\hat{x} \notin dom(\rho)$. But this is not possible because $DFV(e_s) \subseteq dom(\rho)$ by our hypothesis. The second case is that $\rho(\hat{x}) = Va$ and $dsp(C, Va)$ is not defined. But this is not possible because $C, H_s \not\to^\star C', fail$ by our hypothesis, so the $dsp(C, Va)$ is defined and (DVAR) rule applies, contradiction.
- rule (TVAPP) $e_s = \#(e_1, e_2)$
  If e_s is stuck then there are only two cases: (1) e_1 is stuck; (2) e_1 is a value and e_2 is stuck. If e_1 is stuck by induction hypothesis e_1 is a value and by Lemma A.7(3) $e_1 = (x)\{Va\}$. If e_2 reduces, rule (VAAPP2) applies, contradiction. If e_2 is stuck we are in case (2). By induction hypothesis e_2 is a value, hence, rule (VAAPP3) applies, contradiction. If both e_1 and e_2 are values: trivial.
- rule (TDLET) $e_s = dlet\, \hat{x} = e_1\, when\, G\, in\, e_2$
  If e_s is stuck, then it is only the case that e_2 is stuck. By the premise of (TDLET) rule and by Lemma 6.1 $\Gamma \vdash \rho' : K'$ with $\rho' = \rho[G.e, \rho(\hat{x})/\hat{x}]$ and $K' = K_{\hat{x}}, (\hat{x}, \tau, \Delta')$. Since $DFV(e_2) \subseteq DFV(e_s) \subseteq dom(\rho) \subseteq dom(\rho')$ we can apply the induction hypothesis so e_2 is a value. In this case the (DLET2) rule applies, contradiction.

Theorem 6.4 (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 $C, H_s \not\to^+ C', fail$. Then either the computation of e_s terminates yielding a value ($\rho \vdash C, e_s \to^* C'', v$) or it diverges, but it never gets stuck.*

(By contradiction). Assume that $\rho \vdash C, e_s \to^i C'', e^i_s \not\to$ for some $i \in \mathbb{N}$ where e^i_s is a non-value stuck expression. By Proposition 6.3 we have $\Gamma; K \vdash e^i_s : \tau \triangleright H^i_s$ and $C, H_s \to^\star C'', H^i_s$, and since $C, H_s \not\to C', fail$ we have also $C, H^i_s \not\to C', fail$. Then, Theorem 6.2 suffices to show that e^i_s is a value (contradiction). □

PROPERTIES OF THE LOAD TIME ANALYSIS

Specification of the Analysis

In this subsection we prove some properties about the load time analysis in Section 7, in particular we prove Theorems 7.1, 7.2 . Firstly, we define the complete lattice of the analysis estimate by ordering $\wp(Context)$ by inclusion and by exploiting the standard construction of cartesian product and functional space.

Definition A.2. Given two analysis estimates $(\Sigma^1_\circ, \Sigma^1_\bullet)$ and $(\Sigma^2_\circ, \Sigma^2_\bullet)$ we say $(\Sigma^1_\circ, \Sigma^1_\bullet) \sqsubseteq (\Sigma^2_\circ, \Sigma^2_\bullet)$ iff $\Sigma^1_\circ(l) \subseteq \Sigma^2_\circ(l)$ and $\Sigma^1_\bullet(l) \subseteq \Sigma^2_\bullet(l)$ for all $l \in Lab$, where $\subseteq$ is the order of $\wp(Context)$. Furthermore, we define $(\Sigma^1_\circ, \Sigma^1_\bullet) \sqcap (\Sigma^2_\circ, \Sigma^2_\bullet) = (\Sigma^1_\circ \sqcap \Sigma^2_\circ, \Sigma^1_\bullet \sqcap \Sigma^2_\bullet) = (\lambda l.\Sigma^1_\circ(l) \cap \Sigma^2_\circ(l), \lambda l.\Sigma^1_\bullet(l) \cap \Sigma^2_\bullet(l))$.

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

Lemma A.18.

1) *Let be $(\Sigma^\top, \Sigma^\top)$ the top of lattice of the analysis estimates, then $(\Sigma^\top, \Sigma^\top) \vDash H^l$ for all H^l*
2) *Let be $(\Sigma^1_\circ, \Sigma^1_\bullet)$ and $(\Sigma^2_\circ, \Sigma^2_\bullet)$, if $(\Sigma^1_\circ, \Sigma^1_\bullet) \vDash H^l$ and $(\Sigma^2_\circ, \Sigma^2_\bullet) \vDash H^l$ then $(\Sigma^1_\circ \sqcap \Sigma^2_\circ, \Sigma^1_\bullet \sqcap \Sigma^2_\bullet) \vDash H^l$*

Proof. The thesis can be proved by induction on the structure of H^l and by using the analysis rules. The proof is quite standard and below we only discuss the case $H^l = (H^{l_1}_1 \cdot H^{l_2}_2)^l$.

1) By induction hypothesis we know that $(\Sigma^\top, \Sigma^\top) \vDash H^{l_i}_i$ for $i \in \{1, 2\}$. By definition of $\Sigma^\top$, it holds that $\forall l' \in Lab\, \Sigma^\top(l') = Context$. Then, it holds $\Sigma^\top(l) \subseteq \Sigma^\top(l_1)$ and $\Sigma^\top(l_1) \subseteq \Sigma^\top(l_2)$ and $\Sigma^\top(l_2) \subseteq \Sigma^\top(l)$. These inclusions satisfy the premise of the rule (ASEQ1), then we conclude $(\Sigma^\top, \Sigma^\top) \vDash H^l$. The others cases follows the same schema.
2) From hypothesis $(\Sigma^1_\circ, \Sigma^1_\bullet) \vDash H^l$ and the premise of rule (ASEQ1), we know that $\Sigma^1_\circ(l) \subseteq \Sigma^1_\circ(l_1)$, $\Sigma^1_\bullet(l_1) \subseteq \Sigma^1_\circ(l_2)$, $\Sigma^1_\bullet(l_2) \subseteq \Sigma^1_\bullet(l)$ and that $\Sigma^2_\circ(l) \subseteq \Sigma^2_\circ(l_1)$, $\Sigma^2_\bullet(l_1) \subseteq \Sigma^2_\circ(l_2)$, $\Sigma^2_\bullet(l_2) \subseteq \Sigma^2_\bullet(l)$. Since $\cap$ is monotonic with respect $\subseteq$ it holds $\Sigma^1_\circ(l) \cap \Sigma^2_\circ(l) \subseteq \Sigma^1_\circ(l_1) \cap \Sigma^2_\circ(l_1)$, $\Sigma^1_\bullet(l_1) \cap \Sigma^2_\bullet(l_1) \subseteq \Sigma^1_\circ(l_2) \cap \Sigma^2_\circ(l_2)$, $\Sigma^1_\bullet(l_2) \cap \Sigma^2_\bullet(l_2) \subseteq \Sigma^1_\bullet(l) \cap \subseteq \Sigma^2_\bullet(l)$. Then by the induction hypothesis and by the above inclusions we satisfy the premise of rule (ASEQ1) and we conclude $(\Sigma^1_\circ \sqcap \Sigma^2_\circ, \Sigma^1_\bullet \sqcap \Sigma^2_\bullet) \vDash H^l$. □

By exploiting the above two lemmata we can prove

Theorem 7.1 (Existence of solutions)**.** *Given H^l and an initial context C, the set $\{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$ of the acceptable estimates of the analysis for H^l and C is a Moore family; hence, there exists a minimal valid estimate.*

Proof. We need to show that given a set of solutions $Y = \{(\Sigma^i_\circ, \Sigma^i_\bullet) \mid i \in \{1, \ldots, n\}\} \subseteq \{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$, $\sqcap Y \in \{(\Sigma_\circ, \Sigma_\bullet) \mid (\Sigma_\circ, \Sigma_\bullet) \vDash H^l\}$. By applying $n + 1$ times the Lemma A.18 we have that $(\Sigma^\top, \Sigma^\top) \sqcap (\Sigma^1_\circ, \Sigma^1_\bullet) \sqcap \cdots \sqcap (\Sigma^n_\circ, \Sigma^n_\bullet) \vDash H^l$ holds. □

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

Definition A.3 (Immediate subterm)**.** Let H and H_1 be history expressions (for simplicity we ignore labels). We say that H_1 is an immediate subterm of H if $H = H_1 + H_2$, $H = H_2 + H_1$, $H = H_1 \cdot H_2$, $H = H_2 \cdot H_1$, $H = \mu h.H_1$, $H = askG.H_1 \otimes \Delta$.

Lemma A.19 (Pre-substitution)**.** *Let H^l, $H^{l_1}_1$ and $H^{l_2}_2$ be history expressions such that $H^{l_1}_1$ is an immediate subterm of H^l; let $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1}_1$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_2}_2$, for some $(\Sigma_\circ, \Sigma_\bullet)$.*
If $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1)$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l[H^{l_2}_2/H^{l_1}_1]$.

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

- case $\mathfrak{d}$, ϵ^l, $tell\, F^l$, $fail^l$, $retract\, F^l$, h^l
  straightforward
- case $H^l = (H^{l_1}_1 + H^{l_3}_3)^l$
  From the hypothesis and from the premise of rule (ASUM) it holds $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. So by applying the rule (ASUM) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet) \vDash (H^{l_2}_2 + H^{l_3}_3)^l$
- case $H^l = (H^{l_3}_3 + H^{l_1}_1)^l$
  Similar to the previous case.
- case $H^l = (H^{l_1}_1 \cdot H^{l_3}_3)^l$
  From the hypothesis and by the premise of rule (ASEQ1) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l_3)$. So by applying the rule (ASEQ1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet) \vDash (H^{l_2}_2 \cdot H^{l_3}_3)^l$.
- case $H^l = (H^{l_3}_3 \cdot H^{l_1}_1)^l$
  From the hypothesis and by the premise of rule (ASEQ1) we have $\Sigma_\bullet(l_3) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. So by applying the rule (ASEQ1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet) \vDash (H^{l_3}_3 \cdot H^{l_1}_1)^l$.
- case $H^l = (\mu h.H^{l_1}_1)^l$
  From the hypothesis and from the premise of rule (AREC) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. So by applying the rule (AREC) with the new inclusions we have $(\Sigma_\circ, \Sigma_\bullet) \vDash (\mu h.H^{l_2}_2)^l$
- case $H^l = (askG.H^{l_1}_1 \otimes \Delta^{l_3})^l$
  From the hypothesis and from the premise of rule (AASK1) we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. So by the rule (AASK1) with the new inclusions we conclude $(\Sigma_\circ, \Sigma_\bullet) \vDash (askG.H^{l_2}_2 \otimes \Delta^{l_3})^l$ □

Lemma A.20 (Substitution)**.** *Le H^l, $H^{l_1}_1$ and $H^{l_2}_2$ be history expressions such that $H^{l_1}_1$ is a subterm of H^l; let $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1}_1$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_2}_2$, for some $(\Sigma_\circ, \Sigma_\bullet)$.*
If $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l_1)$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l[H^{l_2}_2/H^{l_1}_1]$.

Proof. Since $H^{l_1}_1$ is subterm of H, there exists then another subterm of H, say $H^{l_3}_3$, such that $H^{l_1}_1$ is an immediate subterm of $H^{l_3}_3$. Since $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$, there exists then a subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_3}_3$. Since $H^{l_1}_1$ is an immediate subterm of $H^{l_3}_3$ there exists another subderivation with conclusion $(\Sigma_\circ, \Sigma_\bullet) \vDash H^{l_1}_1$. So by applying

Lemma A.19, we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_3^{l_3}[H_2^{l_2}/H_1^{l_1}]$. Since our analysis is defined on the history expressions syntax and since $\Sigma_\circ(l_3)$ and $\Sigma_\bullet(l_3)$ have not changed, we can reuse the same steps used for $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$ to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l[H_2^{l_2}/H_1^{l_1}]$ □

Theorem 7.2 (Subject Reduction)**.** *Let H^l be a closed history expression such that $(\Sigma_\circ, \Sigma_\bullet) \vDash H^l$.*
If for all $C \in \Sigma_\circ(l)$ such that $C, H^l \to C', H'^{l'}$ then $(\Sigma_\circ, \Sigma_\bullet) \vDash H'^{l'}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l')$ and $\Sigma_\bullet(l') \subseteq \Sigma_\bullet(l)$.

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

- rule (ANIL)
  The statement vacuously holds.
- rule (AASK2)
  The statement vacuously holds.
- rule (AEPS)
  We know that in this case $C, \epsilon^l \to C, \text{ə}$, then the statement vacuously holds.
- rule (ATELL)
  We know that in this case $C, tell\, F^l \to C \cup \{F\}, \text{ə}$, then the statement vacuously holds.
- rule (ARETRACT)
  Similar to (ATELL) rule
- rule (ASEQ1)
  In this case we have $H = (H_1^{l_1} \cdot H_2^{l_2})^l$ and $H' = (H_3^{l_3} \cdot H_2^{l_2})^l$. We have to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash (H_3^{l_3} \cdot H_2^{l_2})^l$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l)$ (trivial) and $\Sigma_\bullet(l) \subseteq \Sigma_\bullet(l)$ (trivial). By (ASEQ1) premise it holds that $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$, $(\Sigma_\circ, \Sigma_\bullet) \vDash H_2^{l_2}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\circ(l)$. By the premise of the semantic rule it holds $C, H_1 \to C', H_3^{l_3}$. The by the induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_3^{l_3}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_3)$ and $\Sigma_\bullet(l_3) \subseteq \Sigma_\bullet(l_1)$. So

  $$\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_3) \implies \Sigma_\circ(l) \subseteq \Sigma_\circ(l_3)$$
  $$\Sigma_\bullet(l_3) \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2) \implies \Sigma_\bullet(l_3) \subseteq \Sigma_\circ(l_2)$$

  Then, by applying (ASEQ1) rule $(\Sigma_\circ, \Sigma_\bullet) \vDash (H_3^{l_3} \cdot H_2^{l_2})^l$ holds.
- rule (ASEQ2)
  In this case we know $H^l = (\text{ə} \cdot H_2^{l_2})^l$ and $H'^{l'} = H_2^{l_2}$. The thesis is straightforward by the premise of (ASEQ2) rule.
- rule (ASUM)
  In this case we have $H^l = (H_1^{l_1} + H_2^{l_2})^l$ and two cases for $H'^{l'}$:
  1) case $H'^{l'} = H_1'^{l_1'}$. By semantic rule we know $C, H_1^{l_1} \to C', H_1'^{l_1'}$, and by induction hypothesis $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1'^{l_1'}$, $\Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_1')$ and $\Sigma_\bullet(l_1') \subseteq \Sigma_\bullet(l_1)$. Since
     $$\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1) \subseteq \Sigma_\circ(l_1') \implies \Sigma_\circ(l) \subseteq \Sigma_\circ(l_1')$$
     $$\Sigma_\bullet(l_1') \subseteq \Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l) \implies \Sigma_\bullet(l_1') \subseteq \Sigma_\circ(l)$$
     the thesis holds.
  2) case $H'^{l'} = H_2'^{l_2'}$. Similar to case (1).
- rule (AASK1)
  In this case we have $H^l = (askG.\, H_1^{l_1} \otimes \Delta^{l_2})^l$ and two cases for $H'^{l'}$:
  1) case $H'^{l'} = H_1^{l_1}$. In this case we know $C \vDash G$ and by the premise of (AASK1) rule we trivially $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$ and $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$.
  2) case $H'^{l'} = \Delta^{l_2}$. Similar to case (1) but we know $C \nvDash G$.
- rule (AREC)
  In this case we know $H^l = (\mu.H_1^{l_1})^l$ and $H'^{l'} = H_1^{l_1}[(\mu.H_1^{l_1})^l/h]$. By rule premise we know $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$ and $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. We have two cases
  1) h does not occur in H_1. In this case the thesis trivially follows since $H_1^{l_1}[(\mu.H_1^{l_1})^l/h] = H_1$.
  2) h occurs n times with labels $l^1, \ldots, l^n$. Since $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$ and since our analysis rules are defined on the syntax of history expressions, there exists a subderivation of $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}$ proof with conclusion $(\Sigma_\circ, \Sigma_\bullet) \vDash h^{l^i}$. By the premise of the rule (AVAR) we know $\Sigma_\circ(l_i) \subseteq \Sigma_\circ(l)$ and $\Sigma_\bullet(l) \subseteq \Sigma_\bullet(l_i)$. So by applying the Lemma A.20 n-times, we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1^{l_1}[(\mu.H_1^{l_1})^l/h^{l^i}]$ for $i \in \{1, \ldots, n\}$ and $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$ and $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$ follow by the premise of rule (AREC).

□

Analysis algorithm

Here, we present some properties about our constraints and analysis algorithm. In particular we prove the Theorems 7.3 and 7.4.

The following lemma establishes the height of our property domain $\wp(Context^* \cup \{\divideontimes\})$. Recall that $Context^*$ is the set of all contexts may be generated from the initial context by a given history expression.

Lemma A.21. *The height of the complete lattice $\wp(Context^* \cup \{\divideontimes\})$ is $\#Context^* + 1$.*

Proof. The longest chains of $\wp(Context^* \cup \{\divideontimes\})$ can be built iteratively from the bottom element $\emptyset$ by adding a element of $Context^* \cup \{\divideontimes\}$ which is not in the previous element of the chain. These kind of chains are $\#Context^* + 1$ in length. □

Definition A.4 (History expression size)**.** Given a history expression H its size is given by the function $size\colon \mathbb{H} \to \mathbb{N}$ inductively defined as

$$size(\text{ə}) = size(\epsilon^l) = size(tell\, F^l) = size(retract\, F^l) = size(fail^l) = size(h^l) = 1$$
$$size((H_1^{l_1} \cdot H_2^{l_2})^l) = size((H_1^{l_1} + H_2^{l_2})^l) = size(H_1^{l_1}) + size(H_2^{l_2}) + 1$$
$$size((ask\, G.H^{l_1} \cdot \Delta^{l_2})^l) = size(H_1^{l_1}) + size(\Delta^{l_2}) + 1$$
$$size((\mu h.H^{l_1})^l) = size(H^{l_1}) + 1$$

The constraints generated by $\mathcal{C}[\_]$ enjoy the following properties. The first one gives us an upper bound to the

size and to the number of variables of left-hand-size of a constraint.

Lemma A.22. *Let H a history expression and let $\mathcal{C}[H]$ be the generated constraints for H. Then for all $E \subseteq X \in \mathcal{C}[H]$ the size of E is at most 3 and the number of variables in E, i.e. $vars(E)$, is at most 2.*

Proof. By Definition 7.5 it is easy to see that the left-hand-size of all generated constraints can be a variable X, a term $X \boxtimes F$ for some fact F and $\boxtimes \in \{\sqcup, \setminus\}$, a term $X \square G$ and a term $X_1 \square G \Rightarrow X_2$ with $\square \in \{\models, \not\models\}$ for some G and $X_1 \neq X_2$. The thesis follows by considering the case $X_1 \square G \Rightarrow X_2$. □

The second property says that the function $\mathcal{C}[\_]$ generates a linear number of constraints with respect to the size of the history expression.

Lemma A.23. *Let $H_p^{l_p}$ be a history expression such that $size(H_p) = n$, then the cardinality of $\mathcal{C}[H]$ is at most $4n + 1$.*

Proof. By induction over the structure of H_p we prove that $\#(\mathcal{C}[H] \setminus \{\{C\} \subseteq \widehat{\Sigma}_\circ(l_p)\})$ is at most $4n$. So the thesis of the lemma trivially follows.

- cases $H = \mathfrak{d}$, $H = \epsilon^l$, $H = h^l$, $H = tell\, F^l$, $H = retract\, F^l$, $H = fail^l$
  In this case $\#\mathcal{C}[H] \in \{0, 1, 2\}$ and $size(H) = 1$, hence, it holds $\#\mathcal{C}[H] \leq 4$.
- case $H = (H_1 \cdot H_2)^l$
  We have $size(H) = n = n_1 + n_2 + 1$ where $n_i = size(H_i)$ for $i \in \{1, 2\}$. By induction hypothesis it holds $\#\mathcal{C}[H_i] \leq 4n_i$ for $i \in \{1, 2\}$. By Definition 7.5 we have
  $\#\mathcal{C}[H] = \#\mathcal{C}[H_1] + \#\mathcal{C}[H_2] + 3 \leq 4n_1 + 4n_2 + 3 \leq 4(n_1 + n_2 + 1) = 4n$
- case $H = (H_1 + H_2)^l$
  Similar to that of $H = (H_1 \cdot H_2)^l$ except for $\#\mathcal{C}[H] = \#\mathcal{C}[H_1] + \#\mathcal{C}[H_2] + 4$.
- case $H = (ask\, G.H \otimes \Delta)^l$
  Similar to that of $H = (H_1 \cdot H_2)^l$ except for $\#\mathcal{C}[H] = \#\mathcal{C}[H] + \#\mathcal{C}[\Delta] + 4$.
- case $H = (\mu h.H_1)^l$
  We have $size(H) = n = n_1 + 1$ where $n_1 = size(H_1)$. By induction hypothesis it holds $\#\mathcal{C}[H_1] \leq 4n_1$. By Definition 7.5 we have
  $\#\mathcal{C}[H] = \#\mathcal{C}[H_1] + 2 \leq 4n_1 + 2 \leq 4(n_1 + 1) = 4n$

□

Theorem 7.3. *Let H be a history expression and let $(\Sigma_\circ, \Sigma_\bullet)$ be an analysis estimate, then*

$$(\Sigma_\circ, \Sigma_\bullet) \models H \iff (\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H]$$

Proof. The proof is in two parts. In the first part, we prove

$$(\Sigma_\circ, \Sigma_\bullet) \models H \implies (\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H]$$

by induction over the structure of history expression.

- case $H = \mathfrak{d}$
  Trivial since $\mathcal{C}[\mathfrak{d}] = \emptyset$ and $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \emptyset$.
- case $H = \epsilon^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definitions 7.6 $[\![\widehat{\Sigma}_\circ(l)]\!](\Sigma_\circ, \Sigma_\bullet) \subseteq [\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ, \Sigma_\bullet)$. The thesis follows from the premise of the rule (AEPS).
- case $H = tell\, F^l$
  We have that $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \sqcup F \subseteq \widehat{\Sigma}_\bullet(l)\}$ is equivalent to $[\![\widehat{\Sigma}_\circ(l) \sqcup F]\!](\Sigma_\circ, \Sigma_\bullet) \subseteq [\![\widehat{\Sigma}_\bullet(l)]\!](\Sigma_\circ, \Sigma_\bullet)$ by Definition 7.6, $\{C \cup \{F\} \mid C \in \Sigma_\circ(l)\} \subseteq \Sigma_\bullet(l)$. The thesis follows from applying the premise of the rule (ATELL).
- case $H = retract\, F^l$
  Similar to the case $H = tell\, F$, substitute $\widehat{\Sigma}_\circ(l) \setminus F$ for $\widehat{\Sigma}_\circ(l) \sqcup F$ and $C \cup \{F\}$ for $C \setminus \{F\}$.
- case $H = (H_1^{l_1} \cdot H_2^{l_2})^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H_1] \cup \mathcal{C}[H_2] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of rule (ASEQ1) we know that it holds $(\Sigma_\circ, \Sigma_\bullet) \models H_i$ for $i = 1, 2$; so by applying the induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H_i]$ for $i = 1, 2$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definitions 7.6 and 7.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)$. These inequalities hold by the premise of the rule (ASEQ1), so the thesis follows.
- case $H = (H_1^{l_1} + H^{l_2})^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\left[H_1^{l_1}\right] \cup \mathcal{C}\left[H_2^{l_2}\right] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of the rule (ASUM) we know $(\Sigma_\circ, \Sigma_\bullet) \models H_i$ for $i = 1, 2$; by applying the induction hypothesis $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}\left[H_i^{l_i}\right]$ holds for $i = 1, 2$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By applying Definitions 7.6 and 7.4, we have the inequalities $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)$. These inequalities are true by the premise of the rule (ASUM).
- case $H = h^l$
  Knowing $\mathbb{K}(h) = (\mu h.H)^{l_1}$ we need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by applying Definitions 7.6 and 7.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$
- case $H = fail^l$
  By Definitions 7.6 and 7.4 $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\divideontimes\} \subseteq \widehat{\Sigma}_\bullet(l)$ holds iff $\{\divideontimes\} \subseteq \Sigma_\bullet(l)$. The thesis follows from the premise of the rule (AASK2).
- case $H = (\mu h.H_1^{l_1})^l$
  We need to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H] \cup \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of the rule (AREC) we know $(\Sigma_\circ, \Sigma_\bullet) \models H_1$ holds, so by applying the induction hypothesis we have that $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \mathcal{C}[H_1]$ holds too. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \models_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by definitions 7.6 and 7.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. These inequalities hold by the premise of the rule (AREC).
- case $H = (ask\, G.H_1^{l_1} \otimes \Delta^{l_2})^l$

We need to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H] \cup \mathcal{C}[\Delta] \cup \{\widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By the premise of the rule (AASK1) we know $(\Sigma_\circ, \Sigma_\bullet) \vDash H$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash \Delta$ hold, so by applying the induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H]$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[\Delta]$. It remains to prove $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$, i.e. by Definitions 7.6 and 7.4 $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$ if at leaf a $C \in \Sigma_\circ(l)$ satisfies G, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_2)$ and $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)$ if at least a $C \in \Sigma_\circ(l)$ does not satisfy G. The thesis follows by the premise of the rule (AASK1).

In the second part, we prove by structural induction over H that

$$(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H] \implies (\Sigma_\circ, \Sigma_\bullet) \vDash H$$

- case $H = \mathfrak{d}$
  trivial since the rule (ANIL) is an axiom.
- case $H = \epsilon^l$
  By our assumption and by Definitions 7.6 and 7.4 we know $\Sigma_\circ(l) \subseteq \Sigma_\bullet(l)$, so by applying the rule (AEPS) we have $(\Sigma_\circ, \Sigma_\bullet) \vDash \epsilon^l$.
- case $H = tell\, F^l$
  By our assumption and by Definitions 7.6 and 7.4 we have $\{C \cup \{F\} \mid C \in \Sigma_\circ(l)\} \subseteq \Sigma_\bullet(l)$. So the premise of the rule (ATELL) is satisfied and it holds $(\Sigma_\circ, \Sigma_\bullet) \vDash tell\, F^l$.
- case $H = retract\, F^l$
  Similar to the case of $H = tell\, F^l$
- case $H = fail^l$
  By our assumption and by Definitions 7.6 and 7.4 we know $\{\divideontimes\} \subseteq \Sigma_\bullet(l)$, so by applying the rule (AASK2) we have $(\Sigma_\circ, \Sigma_\bullet) \vDash fail^l$.
- case $H = h^l$
  By our assumption and by Definitions 7.6 and 7.4 knowing $\mathbb{K}(h) = (\mu h.H)^{l_1}$ we have $\Sigma_\circ(l) \subseteq \Sigma_\bullet(l_1)$ and $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. The premise of the rule (AVAR) thus is satisfied and we conclude $(\Sigma_\circ, \Sigma_\bullet) \vDash h^l$.
- case $H = (H_1^{l_1} \cdot H_2^{l_2})^l$
  By our assumption we know $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H_i]$ for $i = 1, 2$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_i$ for $i = 1, 2$ hold. Also by Definition 7.4 we know that inequalities $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)$ hold. So the premise of the rule (ASEQ1) is satisfied and we conclude the thesis.
- case $H = (H_1^{l_1} + H_2^{l_2})^l$
  By our assumption we know $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H_i]$ for $i = 1, 2$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\bullet(l_2), \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_i$ for $i = 1, 2$ hold. Also by Definition 7.4 we know that inequalities $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$, $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_2)$, $\Sigma_\bullet(l_2) \subseteq \Sigma_\bullet(l)$ hold. So the premise of the rule (ASUM) is satisfied and the thesis holds.
- case $H = (\mu.H_1^{l_1})^l$
  We know that $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H_1]$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l)\}$ hold by our assumption. By induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1$ and by Definition 7.4 we have $\Sigma_\circ(l) \subseteq \Sigma_\circ(l_1)$, $\Sigma_\bullet(l_1) \subseteq \Sigma_\bullet(l)$. The thesis follows by the rule (AREC).
- case $H = (ask\, G.H_1^{l_1} \otimes \Delta^{l_2})^l$
  By our assumption we know $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[H_1]$, $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \mathcal{C}[\Delta]$, $(\Sigma_\circ, \Sigma_\bullet) \vDash_{sc} \{\widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\circ(l) \subseteq \widehat{\Sigma}_\circ(l_1), \widehat{\Sigma}_\circ(l) \vDash G \Rightarrow \widehat{\Sigma}_\bullet(l_1) \subseteq \widehat{\Sigma}_\bullet(l), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\bullet(l) \subseteq \widehat{\Sigma}_\circ(l_2), \widehat{\Sigma}_\circ(l) \nvDash G \Rightarrow \widehat{\Sigma}_\bullet(l_2) \subseteq \widehat{\Sigma}_\bullet(l)\}$. By applying induction hypothesis we have $(\Sigma_\circ, \Sigma_\bullet) \vDash H_1$ and $(\Sigma_\circ, \Sigma_\bullet) \vDash \Delta$. Also by Definition 7.4 we know that inequalities $A_\circ \subseteq \Sigma_\circ(l_1)$, $A_\bullet \subseteq \Sigma_\bullet(l)$ where $A_\circ = \Sigma_\circ(l)$ and $A_\bullet = \Sigma_\bullet(l_1)$ if there exists a $C \in \Sigma_\circ(l)$ otherwise $A_\circ = A_\bullet = \emptyset$; also, $B_\circ \subseteq \Sigma_\circ(l_2)$ and $B_\bullet \subseteq \Sigma_\bullet(l)$ where $B_\circ = \Sigma_\circ(l)$ and $B_\bullet = \Sigma_\bullet(l_2)$ if there exists a $C \in \Sigma_\circ(l)$ otherwise $B_\circ = B_\bullet = \emptyset$. So the premise of the rule (AASK1) is satisfied and the thesis holds.

□

Theorem 7.4. *Let H be a history expression of size n and let h be the height of the complete lattice $\wp(Context^* \cup \{\divideontimes\})$. The algorithm in Figure 12 terminates and computes the minimal solution of the constraints $\mathcal{C}[H]$ in time $O(h \cdot n)$.*

Proof. Since our algorithm is an instance of the general schema displayed in Table 6.1 of [70], the termination and the correctness follows from Lemma 6.4 of [70]. Furthermore, they prove that the time complexity of the general schema is $O(h \cdot M \cdot N)$ where h is the height of the property lattice; M is an upper bound to the size of the left-hand-size of constraints; and N is the number of constraints. In our case we have $M = 3$ by Lemma A.22 and $N = O(n)$ by Lemma A.23, thus the complexity is $O(h \cdot n)$. Moreover, by Lemma A.21 we have that $h = O(\#Context^*)$, hence the overall complexity becomes $O(\#Context^* \cdot n)$. □

As said in Section 7 we conjecture that $\#Context^*$ is a function $f(n)$ of the size of the given history expression and we plan either to compute $f(n)$ or give a good upper bound to its value.